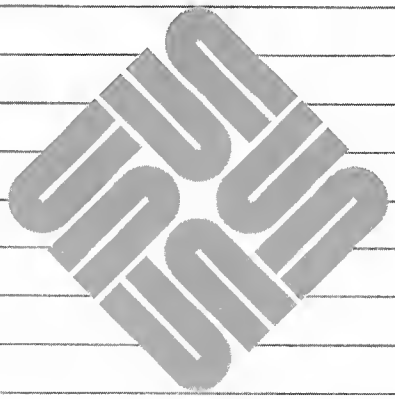
sun microsystems

Network Programming Guide



Part Number: 800-3850-10
Revision A of 27 March, 1990

Trademarks

SunOS™, Sun Workstation®, as well as the word "Sun" followed by a numerical suffix, are trademarks of Sun Microsystems, Incorporated.

ONC is a trademark of Sun Microsystems, Incorporated.

UNIX® and UNIX System V® are trademarks of Bell Laboratories.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Legal Notice to Users

The Network Information Service (NIS) was formerly known as Sun Yellow Pages. The functionality of the two remains the same, only the name has changed. The name Yellow Pages™ is a registered trademark in the United Kingdom of British Telecommunications plc and may not be used without permission.

Contents

Tables

Figures

1

Network Services

This guide gives an overview of the network services available in the Sun 4.1 release. To appreciate the design of these services, it's necessary to see that SunOS is *structurally* a network UNIX system, and is designed to evolve as network technology changes.

SunOS originally diverged from the 4.2BSD UNIX system, a system that already strained at the limits of the UNIX system's original simplicity of design. It was with 4.2BSD that many of the network services found in SunOS were first introduced. Fortunately, the Berkeley designers found alternatives to wedging everything into the kernel. They implemented network services by offloading certain jobs to specialized daemons (server processes) working in close cooperation with the kernel, rather than by adding all new code to the kernel itself. Though NFS is primarily kernel based (using a daemon only to make system calls), SunOS has continued this line of development. Its expanding domain of network services is uniformly built upon a daemon (server) based architecture. Examples of server daemons are the `portmapper`, the network naming service (NIS), the Remote Execution Facility (REX), the Network Lock Manager, and the Status Monitor.

Terminology

A machine that provides resources to the network is called a "server", while a machine that employs these resources is called a "client". A machine may be both a server and a client, and when NFS resources (files and directories) are at issue, often is. A person logged in on a client machine is a "user", while a program or set of programs that run on a client is an "application". There is a distinction between the code implementing the operations of a filesystem, (called "filesystem operations") and the data making up the filesystem's structure and contents (called "filesystem data").

Network services are added to SunOS by means of server processes that are based upon Sun's RPC (Remote Procedure Call) mechanism. These servers are executed on all machines that provide the service. Sun daemons differ significantly from those that were inherited from Berkeley in that most of them are based on RPC. As a consequence, they automatically benefit from the services provided by RPC, and the External Data Representation (XDR) that it is built upon — for example, the data portability provided by XDR and RPC's authentication system.

Anything built with RPC/XDR is automatically a network application, as is anything that stores data in NFS files, even if it doesn't use RPC directly. Furthermore, in so far as network applications can presume the functionality of other

network applications and call upon their services, all network applications are network services as well. The RPC/XDR environment then, is inherently *extensible*. New network services can be easily added by building upon the foundation already in place. In SunOS, then, network services are analogous to UNIX commands — anyone can add one, and when they do they are effectively extending the "system".

NOTE *The term Open Network Computing (ONC) is based on RPC utilities only (such as REX, NIS, Lock Manager, and Status Monitor). The other network utilities described here are not considered part of ONC.*

1.1. The Major Network Services

The *Remote Procedure Call (RPC)* facility is a library of procedures that provide a means whereby one process (the caller process) can have another process (the server process) execute a procedure call, as if the caller process had executed the procedure call in its own address space (as in the local model of a procedure call). Because the caller and the server are now two separate processes, they no longer have to live on the same physical machine.

The *External Data Representation (XDR)*is a specification for the portable data representation standard. RPC uses XDR to ensure that data is represented the same on different computers, operating systems, and computer languages. In SunOS 4.1 XDR is implemented through the socket interface, yet allows programmers to have a standardized access to sockets without being concerned about the low-level details of socket-based IPC.

The *Network File System (NFS)*, is an operating system-independent service which allows users to mount directories, even root directories, across the network, and then to treat those directories as if they were local. There is also an option for a secure mount involving DES authentication of user and host—for more information about it, see the *Secure Networking Features* chapter of *Security Features Guide.*

`portmapper` is a system service upon which all other RPC-based services rely. It's a kind of registrar that keeps track of the correspondence between ports (logical communications channels) and services on a machine, and provides a standard way for a client to look up the port number of any RPC program supported by the server. But in effect, only RPC programs use it.

Sun's *Network Information Service* is a network service designed to ease the job of administering large networks. NIS is a replicated, read-only, distributed database service. Network file system clients use it to access network-wide data in a manner that is entirely independent of the relative locations of the client and the server. The NIS database typically provides password, group, network, and host information.

As part of its System V compatibility program, Sun now supports System-V (SVID) compatible advisory file and record locking for both local and NFS mounted files. User programs simply issue `lockf()` and `fcntl()` system calls to set and test file locks — these calls are then processed by *Network Lock Manager* daemons, which maintain locks at the network level, even in the face of multiple machine crashes.

The lock-manager daemons are able to manage machine crashes because they are based upon a general purpose *Network Status Monitor*. This monitor provides a mechanism by which network applications can detect machine reboots and trigger application-specific recovery mechanisms. The Lock Manager is therefore equipped with a flexible fault-tolerant recovery capability.

There are other network services — NIS and REX[1] are two obvious examples — and there are many others that are certainly services in the broad sense. This section, however, is intended as an introduction, and it covers only the fundamental services noted above.

1.2. Network Programming Manual Overview

This *Network Programming* manual contains this *Network Services* overview and then three major sections. In this overview the fundamental network services are introduced without dealing with any protocol or implementation related issues.

PART ONE focuses on Sun's network programming mechanisms. It includes:

- The `rpcgen` *Programming Guide*, which introduces the `rpcgen` protocol compiler and the C-like language that it uses to specify RPC applications and define network data. In almost all cases, `rpcgen` will allow network applications developers to avoid the use of lower-level RPC mechanisms.
- The *Remote Procedure Call Programming Guide* is intended for programmers who wish to understand the lower-level RPC mechanisms. Readers are assumed to be familiar with the C language and to have a working knowledge of network theory.
- The *External Data Representation: Sun Technical Notes*, which introduces XDR and explains the justification for its "canonical" approach to network data interchange. This section also gives Sun implementation information and a few examples of advanced XDR usage.

PART TWO includes a number of number of protocol specifications. Both the *External Data Representation Protocol Specification* and *Remote Procedure Call Specification* have been published as a DARPA RFC (Request for Comments). These protocol specifications include:

- The *External Data Representation Protocol Specification*, which includes a complete specification of XDR data types, a discussion of the XDR approach and a number of examples of XDR usage. This specification is published as DARPA RFC 1014.
- The *Remote Procedure Call Protocol Specification*, which includes a discussion of the RPC model, a detailed treatment of the RPC authentication facilities and a complete specification of the `portmapper` Protocol. This specification is published as DARPA RFC 1057.

[1] These, however, are not *fundamental* network services, in the same sense as NFS. REX, for example, cannot be guaranteed to be portable to a non-UNIX environment. This is true because the executability of a program depends on many environmental factors — from machine architecture to operating-system services — that are not universally available.

- The *Network File System: Version 2 Protocol Specification*, which includes a complete specification of the Mount Protocol, as well as the NFS specification itself. This specification is published as DARPA RFC 1094.

PART THREE documents Transport-Level Network Programming.

- The first chapter, *Transport Level Interface (TLI) Programming*, describes the TLI system interface for direct access to network mechanisms.

The rest of the chapters in this part document the Berkeley style, socket-Based Inter-Process Communications mechanisms.

- *A Socket-Based Interprocess Communications Tutorial* then introduces socket-based IPC. It assumes little more that basic networking concepts on the part of its reader, and includes many examples.
- *An Advanced Socket-Based Interprocess Communications Tutorial*, which takes up where the *Tutorial* leaves off.
- *Berkeley-Style IPC Implementation Notes*, which describes the low-level networking primitives (e.g. `accept()`, `bind()` and `select()`) which originated with the 4.2BSD UNIX system. This document is of interest primarily to system programmers and aspiring UNIX gurus.

1.3. The Network File System (NFS)

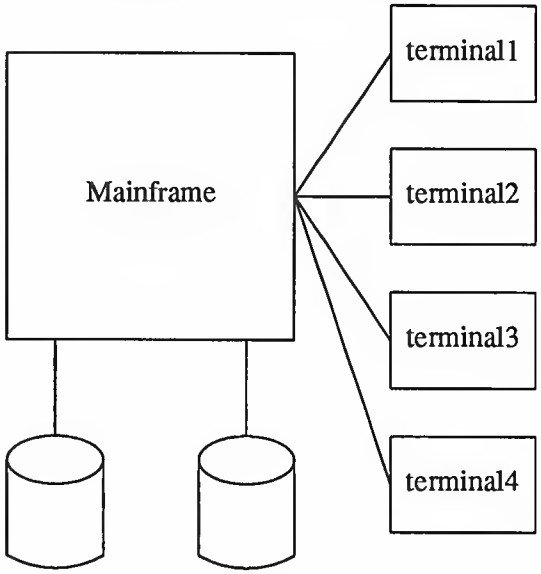
The Network File System is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. Sharing is accomplished by mounting a remote filesystem, then reading or writing files in place.

NFS was *not* designed by extending SunOS onto the network — such an approach was considered unacceptable because it would mean that every computer on the network would have to run SunOS. Instead, operating-system independence was taken as an NFS design goal, along with machine independence, crash recovery, transparent access and high performance. NFS was thus designed as a collection of network services, and not as a distributed operating system. As such, it is able to support distributed applications without restricting the network to a single operating system.

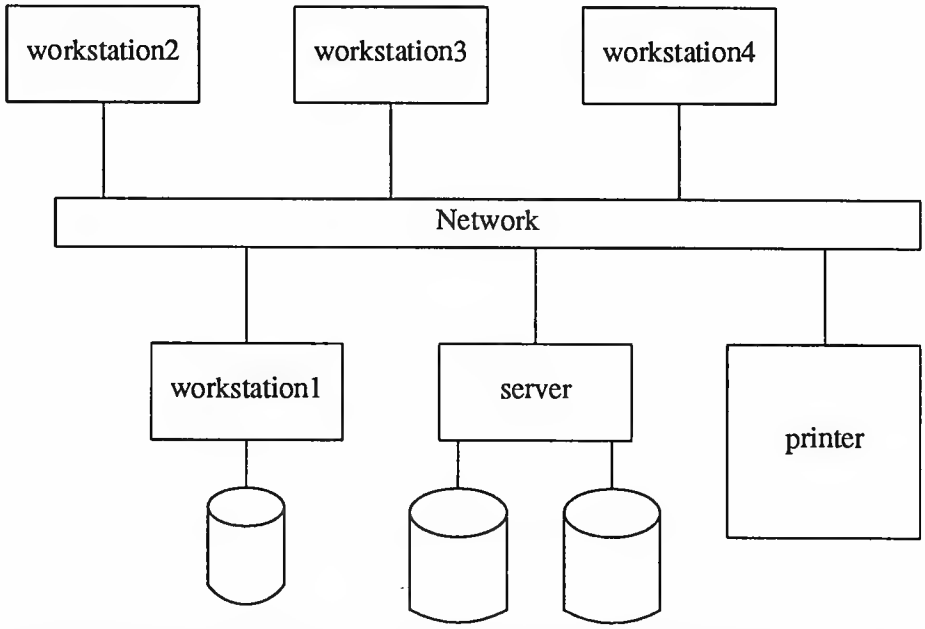
Sun's implementation of NFS is integrated with the SunOS kernel for reasons of efficiency, although such close integration is not strictly necessary. Other vendors will make different choices, as dictated by their operating environments and applications. And because of NFS's open design, all of these applications will be able to work together on a single network.

Computing Environments

The traditional timesharing environment looks like this:



The major problem with this environment is competition for CPU cycles. The workstation environment solves that problem, but requires more disk drives. A network environment looks like this:



The goal of the NFS design was to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network. Printers and supercomputers may also be available somewhere on the network.

Example NFS usage

This section gives three examples of NFS usage.

Example 1: Mounting a Remote Filesystem

Suppose your machine name is `client`, that you want to read some on-line manual pages, and that these pages are not available on your server machine, named `server`, but are available on another machine named `docserv`. Mount the directory containing the manuals as follows:

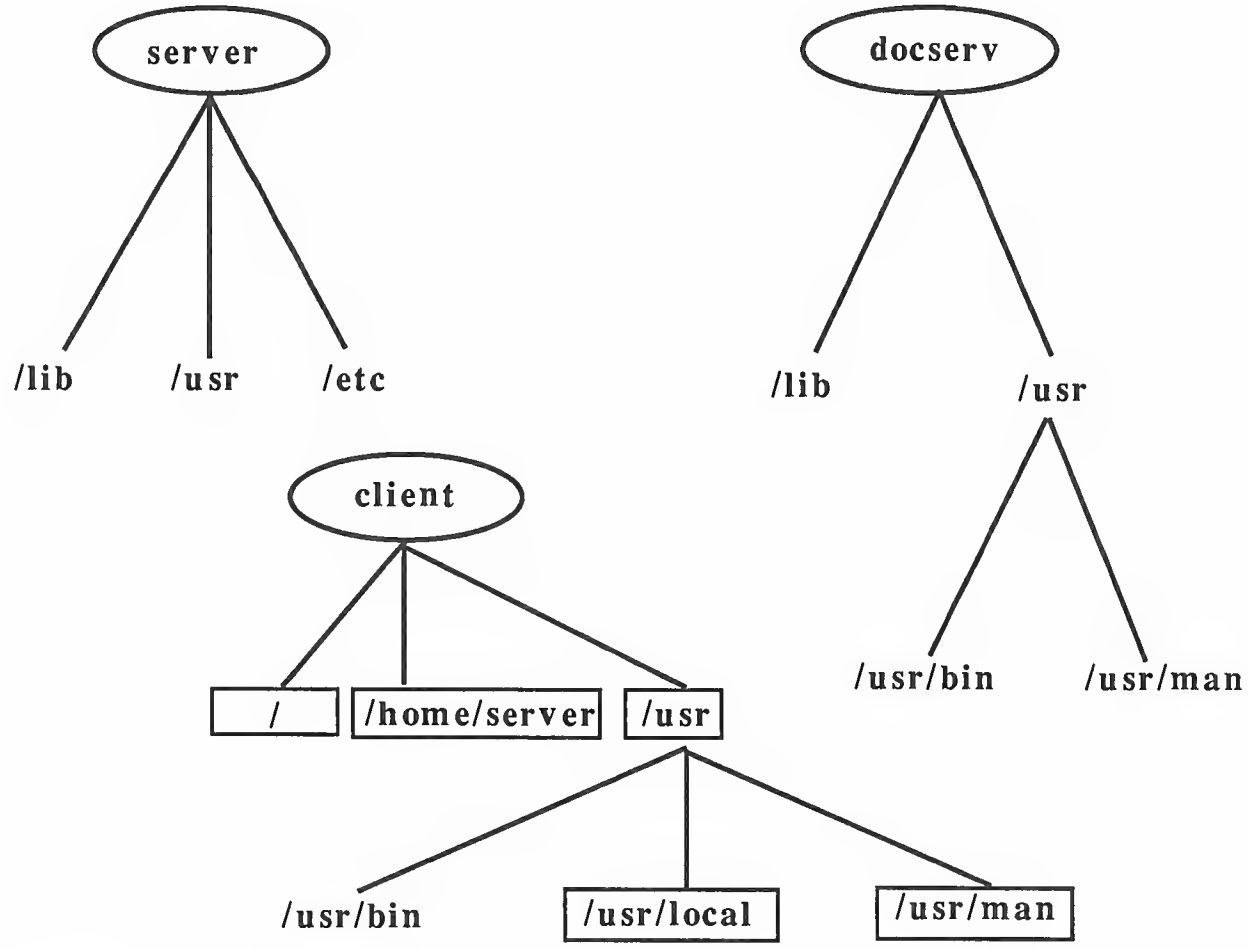
```
client# /usr/etc/mount docserv:/usr/man /usr/man
```

Note that you have to be superuser in order to do this. Now you can use the `man` command whenever you want. Try running the `mount -p` command (on `client`) after you've mounted the remote filesystem. Its output will look something like this:

```
server:/roots/client      /              nfs rw,hard      0 0
server:/usr               /usr           nfs ro           0 0
server:/home/server       /home/server   nfs rw,bg        0 0
server:/usr/local         /usr/local     nfs ro,soft,bg   0 0
docserv:/usr/man          /usr/man       nfs ro,soft,bg   0 0
```

You can remote mount not only filesystems, but also directory hierarchies inside filesystems. In this example, `/usr/man` is not a filesystem mount point — it's just a subdirectory within the `/usr` filesystem. Here's a diagram showing a few key directories of the three machines involved in this example. Ellipses represent machines, and NFS-mounted filesystems are shown boxed. There are five such boxed directories, corresponding to the five lines shown in the `mount -p` output above. The `docserv:/usr/man` directory is shown mounted as the `/usr/man` directory on `client`, as it would be by the `mount` command given above.

Figure 1-1 *An Example NFS Filesystem Hierarchy*



Example 2: Exporting a Filesystem

Suppose that you and a colleague need to work together on a programming project. The source code is on your machine, in the directory `/usr/proj`. It doesn't matter whether your workstation is a diskless node or has a local disk. Suppose that after creating the proper directory your colleague tried to remote mount your directory. Unless you have explicitly exported the directory, your colleague's remote mount will fail with a "permission denied" message.

To export a directory, first become superuser and then edit the `/etc/exports` file. If your colleague is on a machine named `cohort`, then you need to run `exportfs(8)` (after putting this line in `/etc/exports`):

```
/usr/proj       -access=cohort
```

If no explicit access is given for a directory, then the system allows anyone on the network to remote mount your directory. By giving explicit access to `cohort`, you have denied access to others. (For more details about the `/etc/exports`, see the `exports(5)` man page). `mountd`, the NFS mount request server, (see *The NFS Interface*, below) reads the file `/etc/xtab` whenever it receives a request for a remote mount. The file `/etc/xtab` contains the

entries for directories that are currently exported. Now your cohort can remote mount the source directory by issuing this command:

```
cohort# /etc/mount client:/usr/proj /usr/proj
```

This, however, isn't the end of the story, since NFS requests are also checked at request time. If you do nothing, the accesses that you've established in your `/etc/exports` file will stay in effect, but you (and your programs) are free to change them at any time with the `exportfs` command and system call.

Since both you and your colleague will be able to edit files on `/usr/proj`, it would be best to use the `sccs` source code control system for concurrency control.

Example 3: Administering a Server Machine

System administrators must know how to set up the NFS server machine so that client workstations can mount all the necessary filesystems. You export filesystems (that is, make them available) by placing appropriate lines in the `/etc/exports` file. Here is a sample `/etc/exports` file for a typical server machine:

```
/                -access=systems
/exec            -access=engineering:joebob:shilling
/usr             -access=engineering
/home/server     -access=engineering
/home/local.sun2 -access=engineering:athena
/home/local.sun3 -access=engineering
```

Machine names or netgroups, such as `staff` (see `netgroup(5)`) may be specified after the filesystem, in which case remote mounts are limited to machines that are a member of this netgroup. For the complete syntax of the `/etc/exports` file, see `exports(5)`. At any time, the system administrator can see which filesystems are remote mounted by executing the `showmount` command.

NFS Architecture

Transparent Information Access

Users are able to get directly to the files they want without knowing the network address of the data. To the user, all NFS-mounted filesystems look just like private disks. There's no apparent difference between reading or writing a file on a local disk, and reading or writing a file on a disk in the next building. Information on the network is truly distributed.

Different Machines and Operating Systems

No single vendor can supply tools for all the work that needs to get done, so appropriate services must be integrated on a network. NFS provides a flexible, operating system-independent platform for such integration.

Easily Extensible

A distributed system must have an architecture that allows integration of new software technologies without disturbing the extant software environment. Since the NFS network-services approach does not depend on pushing the operating system onto the network, but instead offers an extensible set of protocols for data exchange, it supports the flexible integration of new software.

Ease of Network Administration

The administration of large networks can be complicated and time-consuming, yet they should (ideally) be *at least* as easy to administer as a set of local filesystems on a timesharing system. The UNIX system has a convenient set of maintenance commands developed over the years, and the Network Information Service, a NFS-based network database service, has allowed them to be adapted and extended for the purpose of administering a network of machines. The NIS also allows certain aspects of network administration to be centralized onto a small number of file servers, e.g. only server disks must be backed up in networks of diskless clients. An overview of the NIS facility is presented in the *The Network Information Service Database Service* section of this manual.

The NIS interface is implemented using RPC and XDR, so it is available to non-UNIX operating systems and non-Sun machines. NIS servers do not interpret data, so it is easy for new databases to be added to the NIS service without modifying the servers.

Reliability

NFS's reliability derives from the robustness of the 4.2BSD filesystem, from the stateless NFS protocol[2], and from the daemon-based methodology by which network services like file and record locking are provided. See *The Network Lock Manager* for more details on locking. In addition, the file server protocol is designed so that client workstations can continue to operate even when the server crashes and reboots.

The major advantage of a stateless server is robustness in the face of client, server, or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network gets fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a professional operations staff, and which may be running untested systems that are often rebooted without warning.

High Performance

The flexibility of NFS allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks, and clients with no disks, may yield better performance at lower cost than having many machines with small, inexpensive disks. Furthermore, it is possible to distribute the filesystem data across many servers and get the added benefit of multiprocessing without losing transparency. In the case of read-only files, copies can be kept on several servers to avoid bottlenecks.

Sun has also added several performance enhancements to NFS, such as "fast paths" for key operations, asynchronous service of multiple requests, disk-block caching , and asynchronous read-ahead and write-behind. The fact that caching and read-ahead occur on both client and server effectively increases the cache size and read-ahead distance. Caching and read-ahead do not add state to the server; nothing (except performance) is lost if cached information is thrown

[2] The NFS protocol is *stateless* because each transaction stands on its own. The server doesn't have to remember anything — about clients or files — between transactions.

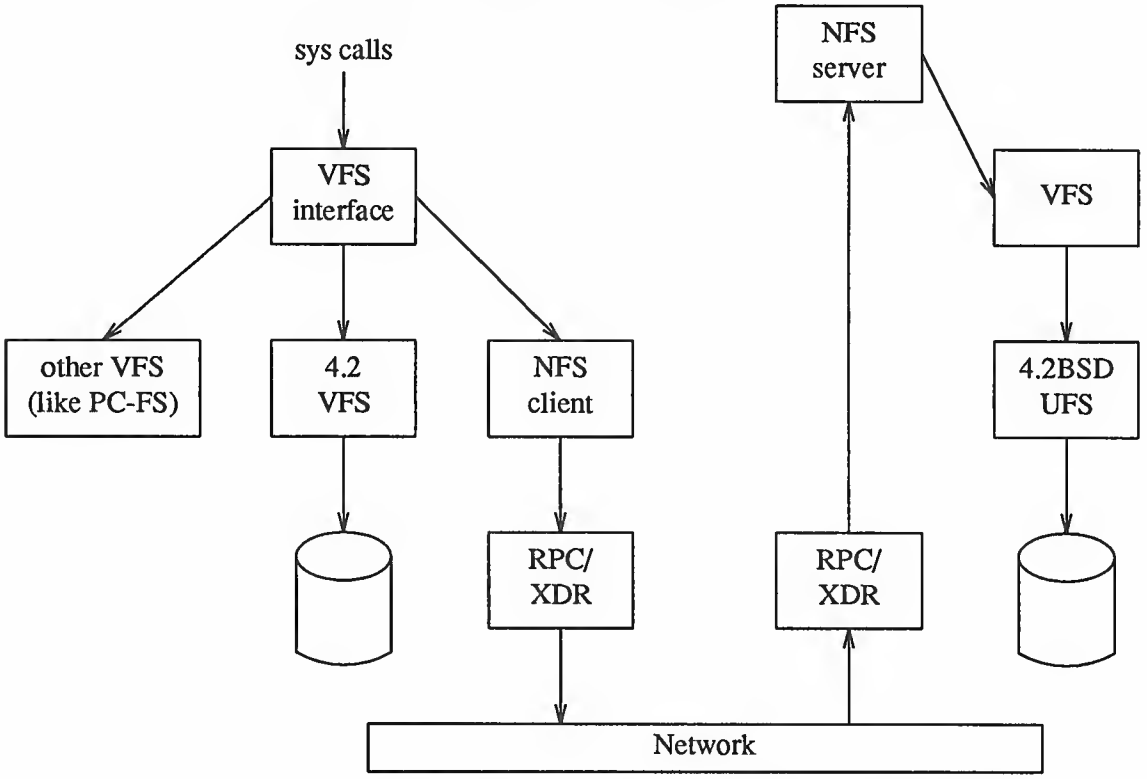
away. In the case of write-behind, both the client and server attempt to flush critical information to disk whenever necessary, to reduce the impact of an unanticipated failure; clients do not free write-behind blocks until the server confirms that the data is written.

The Sun NFS Implementation

In the Sun NFS implementation, there are three entities to be considered: the operating system interface, the *virtual file system (VFS)* , interface, and the network file system (NFS) interface. The UNIX operating system interface has been preserved in the Sun implementation of NFS, thereby insuring compatibility for existing applications. Applications will use `read(2)` and `write(2)` to access NFS files just as the do to access local files.

The VFS is best seen as a layer that Sun has wrapped around the traditional UNIX filesystem. This traditional filesystem is composed of directories and files, each of which has a corresponding `inode` (index node), containing administrative information about the file, such as location, size, ownership, permissions, and access times. Inodes are assigned unique numbers within a filesystem, but a file on one filesystem could have the same number as a file on another filesystem. This is a problem in a network environment, because remote filesystems need to be mounted dynamically, and numbering conflicts would cause havoc. To solve this problem, Sun designed the VFS, which is based on a data structure called a `vnode`. In the VFS, files are guaranteed to have unique numerical designators, even within a network. Vnodes cleanly separate filesystem operations from the semantics of their implementation. Above the VFS interface, the operating system deals in `vnodes`; below this interface, the filesystem may or may not implement `inodes`. The VFS interface can connect the operating system to a variety of filesystems (for example, 4.2 BSD or MS-DOS). A local VFS connects to filesystem data on a local device.

The remote VFS defines and implements the NFS interface on the basis of the RPC and XDR mechanisms. The figure below shows the flow of a request from a client (at the top left) to a collection of filesystems.



In the case of access through a local VFS, requests are directed to filesystem data on devices connected to the client machine. In the case of access through a remote VFS, the request is passed through the RPC and XDR layers onto the net. In the current implementation, Sun uses the UDP/IP protocols and the Ethernet. On the server side, requests are passed through the RPC and XDR layers to an NFS server; the server uses `vnodes` to access one of its local VFSs and service the request. This path is retraced to return results.

Sun's implementation of NFS provides five types of transparency:

1. *Filesystem Type:* The `vnode`, in conjunction with one or more local VFSs (and possibly remote VFSs) permits an operating system (hence client and application) to interface transparently to a variety of filesystem types.
2. *Filesystem Location:* Since there is no differentiation between a local and a remote VFS, the location of filesystem data is transparent.
3. *Operating System Type:* The RPC mechanism allows interconnection of a variety of operating systems on the network, and makes the operating system type of a remote server transparent.
4. *Machine Type:* The XDR definition facility allows a variety of machines to communicate on the network and makes the machine type of a remote server transparent.

5. *Network Type:* RPC and XDR can be implemented for a variety of transport protocols, thereby making the network type transparent.

Simpler NFS implementations are possible at the expense of some advantages of the Sun version. In particular, a client (or server) may be added to the network by implementing one side of the NFS interface. An advantage of the Sun implementation is that the client and server sides can be symmetrical; thus, it is possible for any machine to be client, server, or both. Users at client machines with disks can arrange to share them over NFS without having to appeal to a system administrator or configure a different system on their workstation.

The NFS Interface

As mentioned in the preceding section, a major advantage of NFS is the ability to mix filesystems. In keeping with this, Sun encourages other vendors to develop products to interface with Sun network services. The specifications for RPC and XDR have been placed in the public domain, and Sun's implementation of RPC and XDR is freely licensed, whic serves as a standard for anyone wishing to develop applications for the network. Furthermore, the NFS interface itself is open and can be used by anyone wishing protocol specifications to implement an NFS client or server for the network.

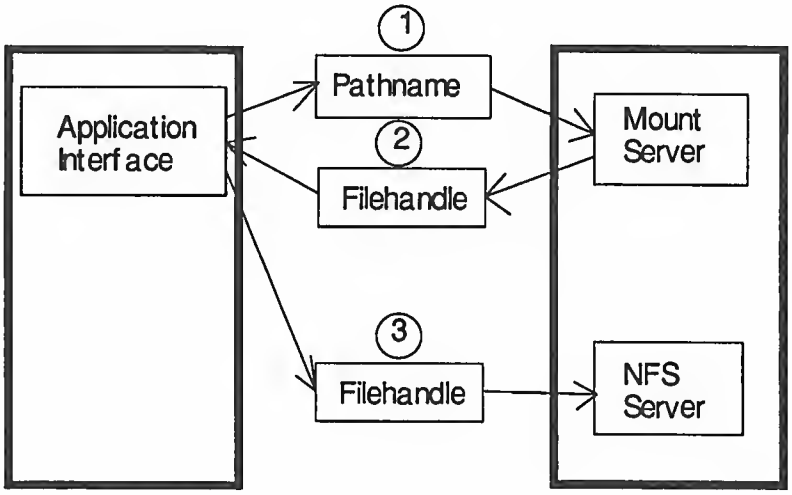
The NFS and the Mount Protocol

The NFS interface defines traditional filesystem operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed so that file operations address files with an uninterpreted identifier called a *filehandle*, a starting byte address, and a length in bytes. NFS *never* deals with pathnames, only with filehandles.

More precisely, NFS never *interprets* pathnames. Some NFS procedures take pathname arguments, but they are just strings to NFS.

Given a filehandle for a directory, a client program can use NFS procedures to get other filehandles and thereby navigate throughout the directories and files of a filesystem. A client must, however, get its first filehandle for a filesystem by using RPC to call the mount server. Mount will return a filehandle that grants access to the filesystem. Figure *1-2* shows the interaction between a client program, a mount server, and an NFS server. Note that the only interface between a mount server and an NFS server is a common filehandle.

Figure 1-2 *Mount and NFS Servers*



Legend:
1. Client sends pathname to mount server
2. Mount server returns corresponding filehandle
3. Client sends filehandle to NFS server

Pathname Parsing

Although many operating systems have analogs to the hierarchical NFS directory and file structure, the conventions used by operating systems to formulate pathnames vary considerably. To accommodate the many possible path naming conventions, the mount procedure is not defined in the NFS protocol but in a separate mount protocol. Actually the mount protocol is the same for any Operating System. It is only the implementation that differs between systems.

The `mount` procedure in the UNIX mount protocol converts a UNIX pathname into a filehandle. If local pathnames can be reasonably mapped to UNIX pathnames, an NFS server developer may wish to implement the UNIX mount protocol, even though the server runs on a different operating system. This approach makes the server immediately usable by clients that use the UNIX protocol and eliminates the need to develop a new `mount` command for UNIX-based clients.

Alternatively, a server developer can obtain a new remote program number from Sun and define a new mount protocol. For example, the `mount` procedure in a VMS Mount protocol would take a VMS file specification rather than a UNIX pathname. Mount protocols are not mutually exclusive; a server could, for example, support the UNIX protocol for UNIX clients and a Multics protocol for Multics clients. Both protocols would return filehandles defined by the NFS implementation on their server.

The mount protocols remove pathname parsing from the NFS protocol, so that a single NFS protocol can work with multiple operating systems. This means that

users and client programs need to know the details of a server's path naming conventions only when mounting a filesystem. Different server path naming conventions therefore typically have little impact on users.

Because mounts are relatively infrequent operations, mount servers can be implemented outside of operating system kernels without materially affecting overall file system performance. Because user-level code is easier to write and far easier to debug than kernel code, mount servers are fairly simple to put together.

Export and Mount Lists

Technically, a mount protocol needs to define only a `mount` procedure that bootstraps the first filehandle for a filesystem. (By convention, a mount protocol should also define a NULL procedure). However, adding other procedures can simplify network management. As a convenience to clients, a mount protocol might provide a procedure that returns a list of filesystems exported by a server. Another useful item is a mount list, a list of clients and the pathnames they have mounted from the server. The UNIX mount protocol defines a mount list and a procedure called `readmount()` that returns the list. With the help of `readmount()`, an administrator can notify the clients of a server that is about to be shut down.

Note that a mount list makes a mount server stateful. Recall, however, that the business of a mount server is to translate pathnames into filehandles; the state represented by a mount list does not affect a server's ability to operate correctly. Neither servers nor clients need take any action to update or rebuild a mount list after a crash. Mount server users should regard the mount and export lists provided by a mount server as "accessories" that are usually, but not necessarily, accurate.

UNIX Mount Protocol Procedures

The mount protocol consists of the six remote procedures listed in Table *1-1*. The `mount()` procedure transforms a UNIX pathname into a filehandle which the client can then pass to the associated NFS server. The pathname passed to the mount procedure usually refers to a directory, often the root directory of a filesystem, but it can name a file instead. In addition to returning the filehandle, `mount` adds the client's host name and the pathname to its mount list. The `readmount()` procedure returns the server's mount list. `unmount()` removes an entry from the server's mount list and `unmountall()` removes all of a client's mount list entries. The `readexport()` procedure returns the server's export list.

Table 1-1 *MOUNT: Remote Procedures, Version 1*

Number	*Name*	*Description*
0	`null`	Do nothing
1	`mount`	Return filehandle for pathname
2	`readmount`	Return mount list
3	`unmount`	Remove mount list entry
4	`unmountall`	Clear mount list
5	`readexport`	Return export list

A Stateless Protocol

The NFS interface is defined so that a server can be *stateless*. This means that a server does not have to remember from one transaction to the next anything about its clients, transactions completed or files operated on. For example, there is no `open()` operation, as this would imply state in the server; of course, the UNIX interface uses an `open()` operation, but the information in the UNIX operation is remembered by the client for use in later NFS operations.

An interesting problem occurs when a UNIX application unlinks an open file. This is done to achieve the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by NFS, the call to `unlink()` will remove the file, since the server does not remember that the file is open. Thus, subsequent operations on the file will fail. In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and unlinks the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files are removed as a part of periodic filesystem maintenance.

Another example of the advantages gained by having the NFS interface to the UNIX system without introducing state is the `mount` command. A UNIX client of NFS "builds" its view of the filesystem on its local devices using the `mount` command or via `automount`; thus, it is natural for the UNIX client to initiate its contact with NFS and build its view of the filesystem on the network with an extended `mount` command. This `mount` command does not imply state in the server, since it only acquires information for the client to establish contact with a server. The `mount` command may be issued at any time, but is typically executed as a part of client initialization. The corresponding `umount` command is only an informative message to the server, but it does change state in the client by modifying its view of the filesystem on the network.

The major advantage of a stateless server is robustness in the face of client, server or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network is fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a professional operations staff and may be

running untested systems and/or may be rebooted without warning.

An NFS server can be a client of another NFS server. However, it is not often that a Sun server will not act as an intermediary between a client and another server. Instead, a client may ask what remote mounts the server has and then attempt to make similar remote mounts. The decision to disallow intermediary servers is based on several factors. First, the existence of an intermediary will impact the performance characteristics of the system; the potential performance implications are so complex that it seems best to require direct communication between a client and server. Second, the existence of an intermediary complicates access control; it is much simpler to require a client and server to establish direct agreements for service. Finally, disallowing intermediaries prevents cycles in the service arrangements; Sun prefers this to detection or avoidance schemes.

NFS currently implements UNIX file protection by making use of the authentication mechanisms built into RPC. This retains transparency for clients and applications that make use of UNIX file protection. Although the RPC definition allows other authentication schemes, their use may have adverse effects on transparency.

Note that NFS, although very UNIX-like, is *not* a UNIX filesystem per se — there are cases in which its behavior differs from that which would be expected of the UNIX system proper:

- The guaranteed APPEND_MODE is the most striking of these differences, for it simply is not supported by NFS.

Note: Network access to devices such as tape drivers is a good idea, but it is best implemented as a separate network service whose requirement for stateful operation is kept separate from network access to files.

- NFS does *not* support device operation over NFS. Support of special files is not stateful because the device operations are carried out locally.
- There are also minor incompatibilities between NFS and UNIX file-system interfaces that are dictated by the very nature of remote NFS mounts. For example, a local NFS daemon simply can't tell that a remote disk partition is full until the remote NFS daemon tells it so. Rather than wait for a positive confirm on every write — a strategy that would impose unacceptable performance problems — the local NFS code caches writes and returns to its caller. If a remote error occurs, it gets reported back as soon as possible, but not as immediately as would a local disk.

File locking and other inherently stateful functionality has been omitted from the base NFS definition. In this way, Sun has been able to preserve a simple, general interface that can be implemented by a wide variety of customers. File locking has been provided as a NFS-compatible network service, and Sun is considering doing the same for other features that inherently imply state and/or distributed synchronization. These features, too, will be kept separate from the base NFS definition. In any case, the open nature of the RPC and NFS interfaces means that customers and users who need stateful or complex features can implement them "beside" NFS.

Note: Miscellaneous Network Operations

Sun supports a small number of miscellaneous networking operations that are useful for temporary inter-host connections, isolated file transfers, and access to non-UNIX systems (e.g. VMS machines on the Internet). These operations include `rcp`, `rlogin`, `rsh`, `ftp`, `telnet`, and `tftp`.

- `rcp` is a remote copy utility program that uses "BSD networking facilities" to copy files from one machine to another. The `rcp` user supplies the path name of a file on a remote machine, and receives a stream of bytes in return. Access control is based on the client's login name and host name.

  The major problem with `rcp` is that it's not transparent to the user, who winds up with a redundant copy of the transferred file. With NFS, by contrast, only one copy of the file is necessary. Another problem is that `rcp` does nothing but copy files. To use it as a model for additional network services would be to introduce a remote command for every regular command: for example, `rdiff` to perform differential file comparisons across machines. By providing for the sharing of filesystems, NFS makes this unnecessary.

`rcp` is useful for NFS servers that you have login access to but not NFS access. Files can copied back and forth, yet you don't need any filesystem mounted.

- `rlogin` allows the user to log into a remote machine, directly accessing both its processor and its mounted file systems. It remains useful in NFS-based networks because, with it, users can directly execute commands on remote machines over the network.

- `rsh` allows the user to execute a command on a remote machine. If no command is specified, `rsh` is equivalent to `rlogin`. Unlike the REX-based `on` command, `rsh` does not copy the users local environment to the remote machine before executing the command. This can be a benefit in situations where exporting your local environment might cause problems.

- `ftp` is very much like `rcp`, in that it supports file copying between machines. However, `ftp` is more general that `rcp`, and is not restricted to copies between two UNIX systems.

- `telnet` communicates with another host using the TELNET protocol. It isn't used much because `rlogin` is the standard mechanism for local inter-host communication. But like `ftp`, `telnet` is useful for non-Unix systems.

- `tftp` is like `ftp`, expect that it is simpler and less reliable. This is because `tftp`'s transfer protocol is very simple; it is less robust that `ftp`'s protocol, and offers fewer options. `tftp` is also used as part of the diskless NFS booting procedure (i.e. netdisk).

1.4. Remote File Sharing (RFS)

Remote File Sharing (RFS) provides a means of viewing files that physically reside on remote machines as if they were on the local machine. Remote files are named using the same conventions as for local files, and all operations on remote files work the same as they do on local files. Like NFS, RFS allows application programs to transparently share files across the network.

NFS, however, is stateless, transactions are independent of each other, and thus no recovery is required when a server or client goes down. RFS, in contrast, supports all UNIX semantics as defined by AT&T. Consequently, it saves state across transactions, and must recover when a server or client goes down.

RFS is used in much the same way as NFS. For both, the user accesses remote files by mounting directories which are made available across the network by server processes running on remote machines. The details do vary, though. Machines using RFS make selected directories available for sharing by *advertising* them. Correspondingly, machines are able to augment their own file trees with the advertised files from other machines. This augmentation is performed by means of a *remote mount*, which is a direct extension of the standard mount operation. Once remote directories have been mounted on the local filesystem, they are functionally part of that filesystem and are accessed in the same way as local directories.

Advertise

To allow other machines to access a directory, its owner must advertise it by using the `adv(8)` command. Once advertised, the directory and all files contained in its subtree are available for sharing by any authorized machine.

Unadvertise

A directory can be unadvertised at any time with the `unadv(8)` command. Unadvertising a directory has no effect on existing mounts of that directory, but future mount requests will fail.

Remote Mounts

RFS extends the `mount(8)` operation to include a remote mount. After a machine has advertised a resource, another machine may remotely mount that resource in its own file tree. For example, to advertise a directory named `/fs1`, the administrator of a server machine would type:

```
example% adv DATA /fs1
```

This makes the `/fs1` subtree available for sharing, and specifies that other machines will use the name `DATA` to refer to it when they mount it. The name `DATA` can be almost any name that would work as a file name as long as it does not contain a period (“.”). See below for the special meaning of the period.

Another machine (a client) gains access to the advertised subtree by mounting the remote subtree on the local directory. The remote `/fs1` is mounted on the local `/fs1` with the command

```
example% mount -d DATA /fs1
```

The `-d` option tells the `mount(8)` command that the resource being mounted is remote.

There is no need for the structures of the client and server file trees to match in any way, or for advertised subtrees to be mounted at the same level on the client

as they occupy on the server. If the client had done the remote mount onto its `/usr` directory, then its references to files under `/usr` would yield files in the server subtree under `/fs1`. A client cannot get to parts of the server file tree that are not within an advertised directory.

Resource Naming

Resource naming is modeled after the DARPA domain naming convention, which has a hierarchically structured name space. A domain in this usage is a name space that may encompass a group of machines and a set of resources advertised by that group of machines.

Resource names are made up of two components separated by a period (“.”). For example, *isl.payroll* might represent a resource called *payroll* in domain *isl*, and *isl.acctp* might represent the machine *acctp* within the same domain. Whether a name specifies a resource or a machine is determined by context; there is no syntactic distinction. If a name is unqualified (i.e., if it contains no periods), the associated domain may (in some cases) be inferred from the context.

A domain’s name space is maintained by a *domain name server*, which insures uniqueness of names within the domain and provides a central location for storing information about the machines and advertised resources in the domain. The `adv(8)`, `unadv(8)`, `mount(8)`, `umount(8)`, and `nsquery(8)` commands use the domain name server as a data base for information about advertised resources, such as their names and the servers that own them.

As described above, each resource is assigned a symbolic name when it is advertised, and the resource is subsequently identified (e.g. with a `mount(8)` command issued on a client) using just the domain name and that symbolic name. Because of this symbolic naming of resources, remote users of resources need not know the actual position of the resources within the server’s file tree, nor even what server within the domain is offering the resource. This location independence simplifies references to resources, and allows for the transparent migration of resources among the machines within a domain (for example, for balancing the load among a set of server machines).

RFS Security Features

RFS contains three security features — client authentication, client authorization, and user and group id mapping.

Client Authentication

This feature associates a password with a client machine so that the identity of a prospective client can be checked before a mount request is serviced. Entry and update of passwords is discussed in the `rfadmin(8)`, `rfstart(8)`, and `rfpasswd(8)` commands.

Client Authorization

RFS provides a means of selectively advertising directories through the `adv(8)` command. For example, if you want to advertise `/usr/private`, but only want to authorize machines *mach1* and *mach2* to mount it, you would issue the command:

```
example% adv PRIVATE /usr/private mach1 mach2
```

Without such a list of machines, the `adv(8)` command puts no restrictions on availability.

One may also choose to advertise a directory read-only by using the `-r` option. Here, a remote mount will only succeed if the mount command also includes the `-r` option.

User and Group Id Mapping

Whenever a user accesses a remote file, that user's permissions must be checked as part of the normal processing of the request (for example, an "open to write" is only valid if the user making the request has write permissions on the file). When accessing a file across two machines, there is no guarantee that the user and group ids on the local machine have the same meaning on the other machine.

Some machines handle this problem by requiring the same numeric ids across machines and expecting the administrators to make sure that the `/etc/passwd` and `/etc/group` files are identical across all machines (at least the entries for all users that access remote files). This approach is conceptually simple, but it is not always feasible in practice, especially in large or already established environments.

RFS, therefore, provides a range of id mapping options through the `idload(8)` command. Id mapping is done by a server machine on all incoming requests, as well as in reporting file ownership ids in response to a request from a client machine (e.g. a `stat(2)` or `fstat(2)`). A client machine maps ids in order to determine the effective user or group id to use in executing a program that is stored on a server and is "set user id" or "set group id".

On each machine, mapping can be set globally, for all remote machines, or on a per-machine basis. All mapping is based on one of two default cases:

Id
: This case maps all incoming ids to *id*, which means that remote users will have the permissions associated with *id* in accessing a server's files. This mapping is the default if no other mapping is specified.

Transparent
: This is a null mapping; remote user and group ids are used locally without change.

These base mappings are augmented by two additional capabilities:

Exclude
: This capability excludes selected ids from the default mapping by mapping them to an otherwise unused id. This capability can be used together with the transparent mapping capability to handle a network where the `/etc/passwd` and `/etc/group` files were identical, but certain permissions (e.g. root) are to be disallowed from remote machines.

Map
: This capability provides arbitrary mapping between remote and local ids that have different name or different numeric values. It can be used with the transparent mapping to handle exceptions to "nearly" identical `/etc/passwd` files.

1.5. The Portmapper

Client programs need a way to find server programs; that is, they need a way to look up and find the *port* numbers of server programs.[3] Network transport services do not provide such a service; they merely provide process-to-process message transfer across a network. A message typically contains a transport address which contains a network number, a host number, and a port number. (A port is a logical communications channel in a host — by waiting on a port, a process receives messages from the network).

How a process waits on a port varies from one operating system to the next, but all provide mechanisms that suspend a process until a message arrives at a port. Thus, messages are not sent across networks to receiving processes, but rather to the ports at which receiving processes wait for messages. The portmapper protocol defines a network service that provides a standard way for clients to look up the port number of any remote program supported by a server.

Port Registration

The portmapper on every host is associated with port number 111. The portmapper is one of the few network services that must have such a well-known and dedicated port. Other network services can be assigned port numbers statically or dynamically so long as they register their ports with their host's portmapper. For example, a server program based on Sun's RPC library typically gets a port number at run time by calling an RPC library procedure. Note that a given network service can be associated with port number 1256 on one server and with port number 885 on another; on a given host, a service can be associated with a different port every time its server program is started. Delegating port-to-remote program mapping to portmappers also automates port number administration.

The portmapper is started automatically whenever a machine is booted. As shown in the *Typical Portmapping Sequence* figure, below, both server programs and client programs call portmapper procedures.[4] To find a remote program's port, a client sends an RPC call message to a server's portmapper; if the remote program is registered with the portmapper, it returns the relevant port number in an RPC reply message. The client program can then send RPC call messages to the remote program's port.

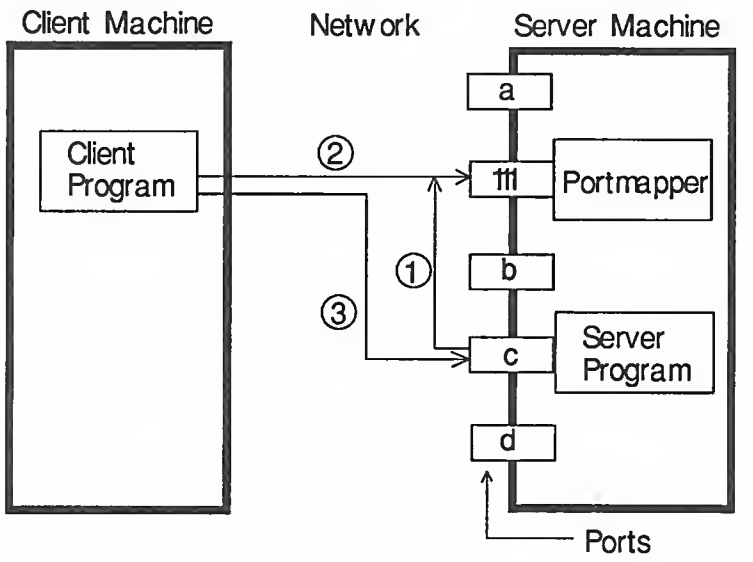
NOTE *The portmapper provides an inherently stateful service because a portmap is a set of associations between registrants and ports. Hence, all the RPC services need to be reregistered if the portmap is restarted.*

---

[3] The naming of services by way of the port-number segment of their IP address is mandated by the Internet protocols. Given this, clients face the problem of determining which ports are associated with the services they wish to use.

[4] Although client and server programs and client and server machines are usually distinct, they need not be. A server program can also be a client program, as when an NFS server calls a portmapper server. Likewise, when a client program directs a "remote" procedure call to its own machine, the machine acts as both client and server.

Figure 1-3 *Typical Portmapping Sequence*



Legend:
1. Server registers with portmapper
2. Client gets server's port from portmapper
3. Client calls server

Note that, because every instance of a remote program can be mapped to a different port on every server, a client has no way to broadcast a remote procedure call directly. However, the portmapper `PMAPPROC_CALLIT` procedure can be used to broadcast a remote procedure call indirectly, since all portmappers are associated with port number 111. One way for a client to find a server running a remote program is to broadcast a call to `PMAPPROC_CALLIT`, asking it to call procedure 0 of the desired remote program.

The Sun RPC library provides an interface to all portmapper procedures. Some of the RPC library procedures also call portmappers automatically on behalf of client and server programs.

1.6. The Network Information Service Database Service

This chapter explains Sun's network database mechanism, the Network Information Service. NIS was previously known as "Yellow Pages", which is now a trademark of British Telecom (refer to the trademark page at the front of this manual). Although it is not intended exclusively for system administrators, it leans towards their concerns. The Network Information Service permits password information and host addresses for an entire network to be held in a single database, and, by so doing, greatly ease system and network administration.

What Is The Network Information Service?

The Network Information Service constitutes a distributed network lookup service:

- NIS is a lookup service: it maintains a set of databases for querying. Programs can ask for the value associated with a particular key, or all the keys, in a database.
- NIS is a network service: programs need not know the location of data, or how it is stored. Instead, they use a network protocol to communicate with a database server that knows those details.
- Network Information Service is distributed: databases are fully replicated on several machines, known as NIS servers. Servers propagate updated databases among themselves, ensuring consistency. At steady state, it doesn't matter which server answers a request; the answer is the same everywhere.

Network Information Service Maps

The Network Information Service serves information stored in NIS *maps*. Each map contains a set of keys and associated values. For example, the `hosts` map contains (as keys) all host names on a network, and (as values) the corresponding Internet addresses. Each NIS map has a *mapname*, used by programs to access data in the map. Programs must know the format of the data in the map. Most maps are derived from ASCII files formerly found in `/etc/passwd`, `/etc/group`, `/etc/hosts`, `/etc/networks`, and other files in `/etc`. The format of data in the NIS map is in most cases identical to the format of the ASCII file. Maps are implemented by `dbm(3X)` files located in subdirectories of `/etc/yp` on NIS server machines.

The relationship between a NIS map and the standard UNIX `/etc` file which it relates to varies from map to map. Some files (e.g. `/etc/hosts`, are *replaced* by their corresponding NIS maps, while some (e.g. `/etc/passwd` are merely *augmented.*

Maps sometimes have nicknames. Although the `ypcat` command is a general NIS database print program, it knows about the standard files in the NIS. Thus `ypcat hosts` is translated into `ypcat hosts.byaddr`, since there is no file called `hosts` in the NIS. The command `ypcat -x` furnishes a list of expanded nicknames.

Network Information Service Domains

A NIS *domain* is a named set of NIS maps. Taken together, these maps define a distinct network namespace and locate a distinct area of administrative control. NIS domains differ from both Internet domains and `sendmail` domains, which define similar kinds of administrative loci in their respective (IP and electronic mail) networks. A given host will typically fall within all three domains, but these domains will not typically coincide. A NIS domain is implemented as a directory in `/etc/yp` containing a set of maps.

You can determine your NIS domain by executing the `domainname` command. A domain name is required for retrieving data from a NIS database. For instance, if your NIS domain is `sys1` and you want to find the Internet address of host `dbserver`, you must ask NIS for the value associated with the key `dbserver` in the map `hosts.byname` within the NIS domain `sys1`. Each machine on the network belongs to a default domain, which is set at boot time. Diskfull

machines have their default domains set by a call to the `domainname` command made from `/etc/rc.local`. Diskless clients have it set as the result of a consultation with the `bootparams(5)` server.

A NIS server holds all the maps of a NIS domain in a subdirectory of `/etc/yp`, named after the domain. In the example above, maps for the `sys1` domain would be held in `/etc/yp/sys1`. A given host can contain maps for more than one NIS domain.

Masters and Slaves

NIS servers containing copies of the same databases can be spread throughout a network. When an arbitrary machine wants information in one of the NIS databases, it makes an RPC call to one of the NIS servers to get it. For any NIS map, one NIS server is designated as the *master* — the only one whose database may be modified. The other NIS servers are *slaves*, and they are automatically updated from time to time to keep their information in sync with that of the master.

All changes to a NIS map must be made on the machine which is the master NIS server for that map. The changes will then propagate to the slaves. A newly built map is timestamped internally when it's created by `makedbm`. If you build a NIS map on a slave server, you will temporarily break the NIS update algorithm, and will have to get all versions in synch manually. Moral: after you decide which server is the master, do all database updates and builds there, not on slaves.

A given server may even be master with regard to one map, and slave with regard to another. This can get confusing quickly. Thus, its recommended that a single server be master for all maps created by `ypinit` in a single domain. Here we are assuming this simple case, in which one server is the master for all maps in a database.

Naming

Imagine a company with two different networks, each of which has its own separate list of hosts and passwords. Within each network, user names, numerical user IDs, and host names are unique. However, there is duplication between the two networks. If these two networks are ever connected, chaos could result. The host name, returned by the `hostname` command and the `gethostname()` system call, may no longer uniquely identify a machine. Thus a new command and system call, `domainname` and `getdomainname()` have been added. In the example above, each of the two networks could be given a different domain name. However, it is always simpler to use a single domain whenever possible.

The relevance of domains to NIS is that data is stored in `/etc/yp/`*domainname*. In particular, a machine can contain data for several different domains.

Data Storage

The data in NIS maps is stored as dbm format databases. (See `dbm(3X)`). Thus the database `hosts.byname` for the domain `sys1` is stored as `/etc/yp/sys1/hosts.byname.pag` and `/etc/yp/sys1/hosts.byname.dir`. The command `makedbm` takes an ASCII file such as `/etc/hosts` and converts it into a dbm file suitable for use by the NIS. However, system administrators normally use the makefile in `/etc/yp` to create new dbm files (read on for details). This makefile in turn calls `makedbm`.

Servers

To become a server, a machine must contain the NIS databases, and must also be running the NIS daemon `ypserv`. The `ypinit` command invokes this daemon automatically. It also takes a flag saying whether you are creating a master or a slave. When updating the master copy of a database, you can force the change to be propagated to all the slaves with the `yppush` command. This pushes the information out to all the slaves. Conversely, from a slave, the `ypxfr` command gets the latest information from the master. The makefile in `/etc/yp` first executes `makedbm` to make a new database, and then calls `yppush` to propagate the change throughout the network.

Clients

Remember that a client machine does not access local copies of `/etc` files, but rather makes an RPC call to a NIS server each time it needs information from a NIS database. NIS clients on NIS servers also don't access local copies of `/etc` files. The `ypbind` daemon remembers the name of a server. When a client boots, `ypbind` broadcasts asking for the name of the NIS server. Similarly, `ypbind` broadcasts asking for the name of a new NIS server if the old server crashes. The `ypwhich` command gives the name of the server that `ypbind` currently points at.

Since client machines don't have entire copies of files in the NIS, the commands `ypcat` and `ypmatch` have been provided. As you might guess, `ypcat passwd` is equivalent to `cat /etc/passwd`. To look for someone's password entry, searching through the password file no longer suffices; you have to issue one of the following commands

```
example% ypcat passwd | grep username
example% ypmatch username passwd
```

where you replace `username` with the login name you're searching for.

Default NIS Files

By default, Sun workstations have a number of files from `/etc` in their NIS: `/etc/passwd`, `/etc/group`, `/etc/hosts`, `/etc/networks`, `/etc/services`, `/etc/protocols`, and `/etc/ethers`. In addition, there is the `netgroup(5)`, file, which defines network wide groups, and used for permission checking when doing remote mounts, remote logins, and remote shells.

In SunOS 4.0, the library routines `getpwent()`, `getgrent()`, and `gethostent()` were rewritten to take advantage of the NIS. Thus, C programs that call these library routines may have to be relinked in order to function correctly.

Hosts

The hosts file is stored as two different NIS maps. The first, `hosts.byname`, is indexed by hostname. The second, `hosts.byaddr`, is indexed by Internet address. Remember that this actually expands into four files, with suffixes `.pag`, and `.dir`. When a user program calls the library routine `gethostbyname()`, a single RPC call to a server retrieves the entry from the `hosts.byname` file. Similarly, `gethostbyaddr()` retrieves the entry from the `hosts.byaddr` file. If the NIS is not running (which is caused by commenting `ypbind` out of the `/etc/rc` file), then `gethostbyname()` will read the `/etc/hosts` files, just as it always has.

Normally, the hosts file for the NIS will be the same as the `/etc/hosts` file on the machine serving as a NIS master. In this case, the makefile in `/etc/yp` will check to see if `/etc/hosts` is newer than the dbm file. If it is, it will use a simple `sed` script to recreate `hosts.byname` and `hosts.byaddr`, run them through `makedbm` and then call `yppush` See `ypmake` for details.

Passwd

The `passwd` file is similar to the `hosts` file. It exists as two separate files, `passwd.byname` and `passwd.byuid`. The `ypcat` program prints it, and `ypmake` updates it. However, if `getpwent` always went directly to the NIS as does `gethostent`, then everyone would be forced to have an identical password file. Consequently, `getpwent` reads the local `/etc/passwd` file, just as it always did. But now it interprets "+" entries in the password file to mean, interpolate entries from the NIS database. If you wrote a simple program using `getpwent` to print out all the entries from your password file, it would print out a virtual password file: rather than printing out + signs, it would print out whatever entries the local password file included from the NIS database.

Others

Of the other files in `/etc`, `/etc/group` is treated like `/etc/passwd`, in that `getgrent()` will only consult the NIS if explicitly told to do so by the `/etc/group` file. The files `/etc/networks`, `/etc/services`, `/etc/protocols`, `/etc/ethers`, and `/etc/netgroup` are treated like `/etc/hosts`: for these files, the library routines go directly to the NIS, without consulting the local files.

Changing your passwd

To change data in the NIS, the system administrator must log into the master machine, and edit databases there; `ypwhich -m` tells where the master server is. However, since changing a password is so commonly done, the `yppasswd` command has been provided to change your NIS password. It has the same user interface as the `passwd` command. This command will only work if the `yppasswdd` server has been started up on the NIS master server machine.

1.7. The Network Lock Manager

SunOS includes an NFS-compatible Network Lock Manager (see the *lockd(8C)* man page for more details) that supports the `lockf()/fcntl()`, System V style of advisory file and record locking over the network. System V locks are generally considered superior to 4.3BSD locks, implemented with the `flock()` system call, for they provide record level, and not merely file level, locking. Record level locking is essential for database systems. Sun does support `flock()` for use on individual machines, but `flock()` is not intended to be used across the network. `flock()` locks exclude only other processes on the

same machine. There is no interaction between `flock()` and `lockf()`.

Locking prevents multiple processes from modifying the same file at the same time, and allows cooperating processes to synchronize access to shared files. The user interfaces with Sun's network locking service by way of the standard `lockf()` system-call interface, and rarely requires any detailed knowledge of how it works. The kernel maps user calls to `flock()` and `fcntl()` into RPC-based messages to the local lock manager (or, if the files in question are on RFS-mounted filesystems, into calls to RFS). The fact that the file system may be spread across multiple machines is really not a complication — until a crash occurs.

All computers crash from time to time, and in an NFS environment, where multiple machines can have access to the same file at the same time, the process of recovering from a crash is necessarily more complex than in a non-network environment. Furthermore, locking is *inherently stateful*. If a server crashes, clients with locked files must be able to recover their locks. If a client crashes, its servers must have the sense to hold the client's locks while it recovers. And, to preserve NFS's overall transparency, the recovery of lost locks must not require the intervention of the applications themselves. This is accomplished as follows:

- Basic file access operations, such as read and write, use a stateless protocol (the NFS protocol). All interactions between NFS servers and clients are atomic — the server doesn't remember anything about its clients from one interaction to the next. In the case of a server crash, client applications will simply sleep until it comes back up and their NFS operations can complete.
- *Stateful services* (those that require the server to maintain client information from one transaction to the next) such as the locking service, are not part of NFS per se. They are separate services that use the status monitor (see *The Network Status Monitor*) to ensure that their implicit network state information remains consistent with the real state of the network. There are two specific state-related problems involved in providing locking in a network context:

  1) if the client has crashed, the lock can be held forever by the server

  2) if the server has crashed, it loses its state (including all its lock information) when it recovers.

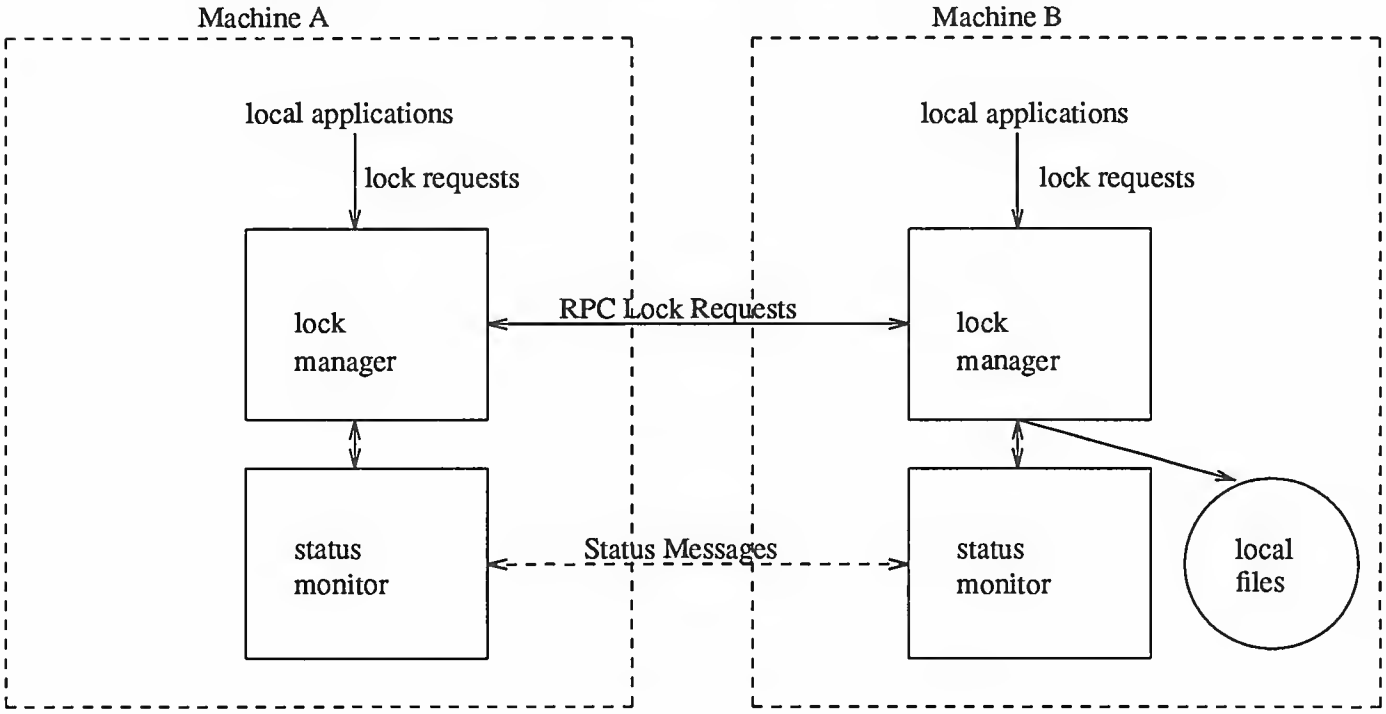
  The Network Lock Manager solves both of these problems by cooperating with the Network Status Monitor to ensure that it's notified of relevant machine crashes. Its own protocol then allows it to recover the lock information it needs when crashed machines recover.

The lock manager and the status monitor are both network-service daemons — they run at user level, but they are essential to the kernel's ability to provide fundamental network services, and they are therefore run on all network machines. They are best seen as extensions to the kernel which, for reasons of space, efficiency and organization, are implemented as daemons. Most application programs will request the network service through a system call to the kernel (like `lockf()`), though it is possible to interact with the service directly with RPC.

With `lockf()` the kernel uses RPC to call the daemon. The network daemons communicate among themselves with RPC (see *The Locking Protocol* for some details of the lock manager protocol). It should be noted that the daemon-based approach to network services allows for tailoring by users who need customized services.

The following figure depicts the overall architecture of the locking service.

Figure 1-4 *Architecture of the NFS Locking Service*



At each server site, a lock manager process accepts lock requests, made on behalf of client processes by a remote lock manager, or on behalf of local processes by the kernel. The client and server lock managers communicate with RPC calls. Upon receiving a lock request for a machine that it doesn't already hold a lock on, the lock manager registers its interest in that machine with the local status monitor, and waits for that monitor to notify it that the machine is up. The monitor continues to watch the status of registered machines, and notifies the lock manager is one of them is rebooted (after a crash). If the lock request is for a local file, the lock manager tries to satisfy it, and communicates back to the application along the appropriate RPC path.

The crash recovery procedure is very simple. If the failure of a client is detected, the server releases the failed client's locks, on the assumption that the client application will request locks again as needed. If the recovery (and, by implication, the crash) of a server is detected, the client lock manager retransmits all lock requests previously granted by the recovered server. This retransmitted information is used by the server to reconstruct its locking state. See below for more details.

The locking service recovers from failure in a stateless manner. Its state information is carefully circumscribed within a pair of system daemons that are set up for automatic, application-transparent crash recovery. If a server crashes, and thus loses its state, it expects that its clients will be notified of the crash and send it the information that it needs to reconstruct its state. The key in this approach is the status monitor, which the lock manager uses to detect both client and server failures.

The Locking Protocol

The lock style implemented by the network lock manager is that specified in the *AT&T System V Interface Definition*, (see the `lockf(2)` and `fcntl(2)` man pages for details). There is no interaction between the lock manager's locks and `flock()`-style locks, which remain supported, but which should be used for non-network applications only.

Locks are presently advisory only, on the (well supported) assumption that cooperating processes can do whatever they wish without mandatory locks. (See the `fcntl(2)` man page for more information about advisory locks).

There are four basic Lock Manager requests that are made by the kernel in response to various `ioctl()/fcntl()` calls:

`KLM_LOCK`
: Lock the specified record.

`KLM_UNLOCK`
: Unlock the specified record.

`KLM_TEST`
: Test if the specified record is locked.

`KLM_CANCEL`
: Cancel an outstanding lock request.

Despite the fact that the network lock manager adheres to the `lockf()/fcntl()` semantics, there are a few subtle points about its behavior that deserve mention. These arise directly from the nature of the network:

- The first and most important of these has to do with crashes. When an NFS-client goes down, the lock managers on all of its servers are notified by their status monitors, and they simply releases its locks, on the assumption that it will request them again when it wants them. When a server crashes, however, matters are different: the clients will wait for it to come back up, and when it does, its lock manager will give the client lock managers a grace period to submit lock reclaim requests, and during this period will accept only reclaim requests. The client status monitors will notify their respective lock managers when the server recovers. The default grace period is 45 seconds.
- It is possible that, after a server crash, a client will not be able to recover a lock that it had on a file on that server. This can happen for the simple reason that another process may have beaten the recovering application process to the lock. In this case the `SIGLOST` signal will be sent to the process (the default action for this signal is to kill the application).

- The local lock manager does not reply to the kernel lock request until the server lock manager has gotten back to it. Further, if the lock request is on a server new to the local lock manager, the lock manager registers its interest in that server with the local status monitor and waits for its reply. Thus, if either the status monitor or the server's lock manager are unavailable, the reply to a lock request for remote data is delayed until it becomes available.

1.8. The Network Status Monitor

The Network Status Monitor (see the *statd(8C)* man page for more details) was introduced with the lock manager, which relies heavily on it to maintain the inherently stateful locking service within the stateless NFS environment. However, the status monitor is very general, and can also be used to support other kinds of stateful network services and applications. Normally, crash recovery is one of the most difficult aspects of network application development, and requires a major design and installation effort. The status monitor makes it more or less routine.

It is anticipated that, in the future, new network services, some of them stateful, will be introduced into the Sun system. These services will use the status monitor to keep up with the state of the network and to cope with machine crashes.

The status monitor works by providing a general framework for collecting network status information. Implemented as a daemon that runs on all network machines, it implements a simple protocol which allows applications to easily monitor the status of other machines. Its use improves overall robustness, and avoids situations in which applications running of different machines (or even on the same machine) come to disagree about the status of a site — a potentially dangerous situation that can lead to inconsistencies in many applications.

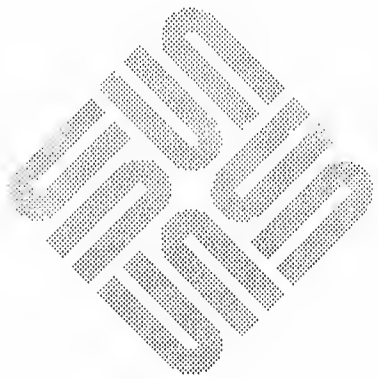
Applications using the status monitor do so by registering with it the machines that they are interested in. The monitor then tracks the status of those machines, and when one of them crashes[5] it notifies the interested applications to that effect, and they then take whatever actions are necessary to reestablish a consistent state.

There are several major advantages to this approach:

- Only applications that use stateful services must pay the overhead — in time and in code — of dealing with the status monitor.
- The implementation of stateful network applications is eased, since the status monitor shields application developers from the complexity of the network.

[5] Actually, when one of them recovers from a crash.

PART ONE: Network Programming



2

Introduction to Remote Procedure Calls

2.1. Overview

What are Remote Procedure Calls? Simply put, they are a high-level communications paradigm which allows network applications to be developed by way of specialized kinds of procedure calls designed to hide the details of the underlying networking mechanisms.

RPC implements a logical client to server communications system designed specifically for the support of network applications. With RPC, the client makes a procedure call which sends requests to the server as necessary. When these requests arrive, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

How it is useful

The net effect of programming with RPC is that programs are designed to run within a client/server network model. Such programs use RPC mechanisms to avoid the details of interfacing to the network, and provide network services to their callers without even requiring that they be aware of the existence and function of the underlying network.

This mechanism solves the tedious issues of programming by making the calls transparent. For example, a program can simply make a call to `rnusers()`, a C routine which returns the number of users on a remote machine. The caller is not explicitly aware of using RPC — they simply call a procedure, much like making a system call to `malloc()`.

Even though this discussion only mentions the interface to C, Remote Procedure Calls can be made from any language. Additionally even though this discussion refers to RPC only as it is used to communicate between processes on different machines, it also works for communication between different processes on the same machine.

Terminology

This chapter discusses servers, services, programs, procedures, clients, and versions. A server provides network services and a network service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification. Network clients initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file IO and have procedures like

"read" and "write". A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

The RPC Model

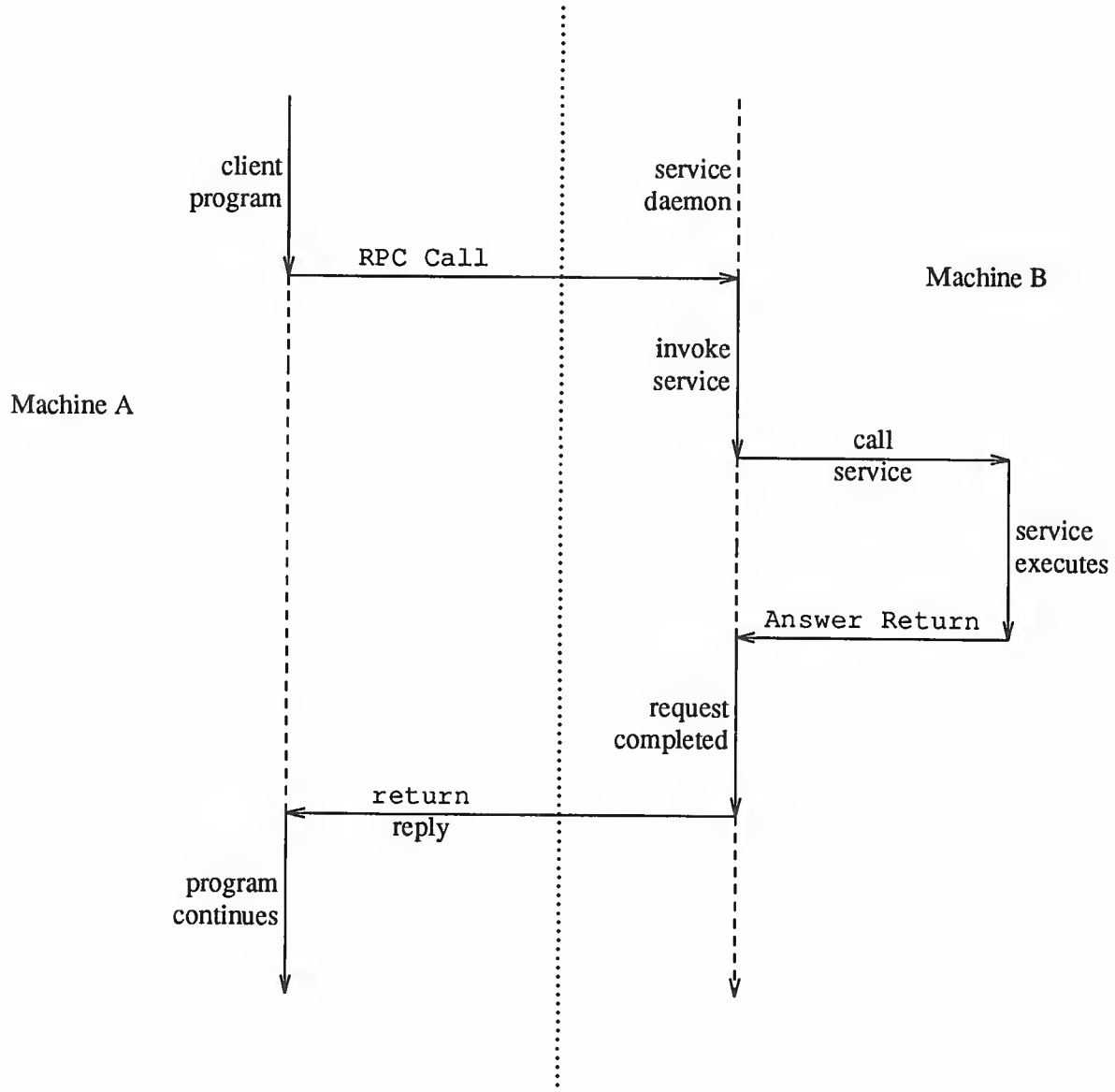
The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes—one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Please refer to Figure *2-1*.

Note that in this model, only one of the two processes is active at any given time. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests. For a more detailed discussion on the RPC protocol, see Chapter *7 — Remote Procedure Calls: Protocol Specification.*

Figure 2-1 *Network Communication with the Remote Procedure Call*



In the above diagram, the details of the network transport are hidden within the Remote Procedure Call. Note, however, that the RPC would not be very useful if those details were entirely unavailable to user and programmers who required access to them.

2.2. Versions and Numbers

Each RPC procedure is uniquely defined by a program number and procedure number. The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. For example, when you want to call a procedure to find the number of remote users, you look up the appropriate program, version and procedure numbers in a

manual, just as you look up the name of a memory allocator when you want to allocate memory.

2.3. Portmap

The `portmap` is the only network service that must have such a well-known (dedicated) port. Other network services can be assigned port numbers statically or dynamically so long as they register their ports with their host's `portmap`. The `portmap` is started automatically whenever a machine is booted. As part of its initialization, a server program calls its host's `portmap` to create a portmap entry for its program and version number. To find a remote program's port, a client sends an RPC call message to a server's `portmap`; if the remote program is registered with the `portmap`, it returns the relevant port number in an RPC reply message. The client program can then send RPC call messages to the remote program's port.

2.4. Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

Transport Selection

Sun RPC is currently supported on both UDP/IP and TCP/IP transports. The selection of the transport depends upon the requirements of the application. UDP (connection less) may be the transport of choice if the application has all of the following characteristics:

1. The procedures are idempotent. i.e. the same procedure can be executed more than once without any harmful side-effects. For example, reading a block of data is idempotent, while creating a file is a non-idempotent operation.

2. The size of both the arguments and results is smaller than the UDP packet size (8 Kbytes for Sun UDP implementation).
3. The server is required to handle many (hundreds) of clients. Since the UDP server does not keeps any state about the client, it can potentially handle many clients. On the other hand, TCP server keeps state for each open client connection and hence the number of clients is limited by the machine resources.

TCP (connection oriented) may be the transport of choice if the application has any of the following requirements and characteristics:

1. The application needs to maintain a high degree of reliability.
2. The procedures are non-idempotent and at-most-once semantics are required.
3. The size of either the arguments or the results exceeds 8 Kbytes.

2.5. External Data Representation

RPC presumes the existence of the eXternal Data Representation (XDR), a standard for the machine-independent description and encoding of data. XDR is useful for transferring data between different computer architectures, and has been used to communicate data between such divers machines as the Sun Workstation, VAX, IBM-PC, and Cray.

RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. For a detailed discussion of XDR, see Chapter *6 — External Data Representation Standard: Protocol Specification.*

2.6. rpcinfo

`rpcinfo` is a command that reports current RPC registration information known to `portmap` (and can be used by administrators to delete registrations). `rpcinfo` can be used to find all the RPC services registered on a specified host and to report their port numbers and the transports for which they are registered. It can also be used to call (ping) a specific version of a specific program on a specific host using TCP or UDP transport, and to report whether the response was received. For details, see the `rpcinfo(8C)` manual pages.

2.7. Assigning Program Numbers

Program numbers are assigned in groups of `0x20000000` according to the following chart:

```
       0x0 - 0x1fffffff Defined by Sun
0x20000000 - 0x3fffffff Defined by user
0x40000000 - 0x5fffffff Transient
0x60000000 - 0x7fffffff Reserved
0x80000000 - 0x9fffffff Reserved
0xa0000000 - 0xbfffffff Reserved
0xc0000000 - 0xdfffffff Reserved
0xe0000000 - 0xffffffff Reserved
```

Sun Microsystems administers the first group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

To register a protocol specification, send a request by network mail to `rpc@sun.com`, or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Please include a compilable `rpcgen ".x"` file describing your protocol. You will be given a unique program number in return.

Some of the RPC program numbers can be found in /etc/rpc. Protocol specifications of standard Sun RPC services can be found in the include files in `/usr/include/rpcsvc`. These services, however, constitute only a small subset of those which have been registered. A list of some of the registered programs is:

Table 2-1 *Registered RPC Program Numbers*

RPC Number	*Program*	*Description*
100000	PMAPPROG	*portmap*
100001	RSTATPROG	*remote stats*
100002	RUSERSPROG	*remote users*
100003	NFSPROG	*nfs*
100004	YPPROG	*NIS*
100005	MOUNTPROG	*mount daemon*
100006	DBXPROG	*remote dbx*
100007	YPBINDPROG	*NIS binder*
100008	WALLPROG	*shutdown msg*
100009	YPPASSWDPROG	*yppasswd server*
100010	ETHERSTATPROG	*ether stats*
100011	RQUOTAPROG	*disk quotas*
100012	SPRAYPROG	*spray packets*
100013	IBM3270PROG	*3270 mapper*
100014	IBMRJEPROG	*RJE mapper*
100015	SELNSVCPROG	*selection service*
100016	RDATABASEPROG	*remote database access*
100017	REXECPROG	*remote execution*
100018	ALICEPROG	*Alice Office Automation*
100019	SCHEDPROG	*scheduling service*
100020	LOCKPROG	*local lock manager*
100021	NETLOCKPROG	*network lock manager*
100022	X25PROG	*x.25 inr protocol*

Table 2-1 *Registered RPC Program Numbers— Continued*

RPC Number	*Program*	*Description*
100023	STATMON1PROG	*status monitor 1*
100024	STATMON2PROG	*status monitor 2*
100025	SELNLIBPROG	*selection library*
100026	BOOTPARAMPROG	*boot parameters service*
100027	MAZEPROG	*mazewars game*
100028	YPUPDATEPROG	*NIS update*
100029	KEYSERVEPROG	*key server*
100030	SECURECMDPROG	*secure login*
100031	NETFWDIPROG	*nfs net forwarder init*
100032	NETFWDTPROG	*nfs net forwarder trans*
100033	SUNLINKMAP_PROG	*sunlink MAP*
100034	NETMONPROG	*network monitor*
100035	DBASEPROG	*lightweight database*
100036	PWDAUTHPROG	*password authorization*
100037	TFSPROG	*translucent file svc*
100038	NSEPROG	*nse server*
100039	NSE_ACTIVATE_PROG	*nse activate daemon*
100043	SHOWHFD	*showfh*
150001	PCNFSDPROG	*pc passwd authorization*
200000	PYRAMIDLOCKINGPROG	*Pyramid-locking*
200001	PYRAMIDSYS5	*Pyramid-sys5*
200002	CADDS_IMAGE	*CV cadds_image*
300001	ADT_RFLOCKPROG	*ADT file locking*

3

rpcgen Programming Guide

3.1. The rpcgen Protocol Compiler

The details of programming applications to use Remote Procedure Calls can be tedious. One of the more difficult areas is writing XDR routines to convert procedure arguments and results into their network format and vice-versa.

Fortunately, `rpcgen(1)` exists to help programmers write RPC applications simply and directly. `rpcgen` does most of the dirty work, allowing programmers to debug the main features of their application, instead of requiring them to spend most of their time on their network interface code.

`rpcgen` is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output for RPC programs. This output includes skeleton versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, a header file that contains common definitions and, optionally, dispatch tables which the server can use to check authorizations and then invoke service routines. The client skeletons' interface with the RPC library and effectively hide the network from their callers. The server skeleton similarly hides the network from the server procedures that are to be invoked by remote clients. `rpcgen`'s output files can be compiled and linked in the usual way. The server code generated by `rpcgen` has support for `inetd` i.e. the server can be started via `inetd` or at the command line.

The developer writes server procedures—in any language that observes system calling conventions—and links them with the server skeleton produced by `rpcgen` to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client skeletons. Linking this program with `rpcgen`'s skeletons creates an executable program. `rpcgen` options can be used to suppress skeleton generation and to specify the transport to be used by the server skeleton.

Like all compilers, `rpcgen` reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including `rpcgen`, do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. `rpcgen` is no exception. In speed-critical applications, hand-written routines can be linked with the `rpcgen` output without any difficulty. Also, one may proceed by using `rpcgen` output as a starting point, and then rewriting it as necessary. (For a discussion of RPC programming without `rpcgen`, see the next chapter, the *Remote Procedure Call Programming Guide*).

Converting Local Procedures into Remote Procedures

Assume an application that runs on a single machine, one which we want to convert to run over the network. Here we will demonstrate such a conversion by way of a simple example—a program that prints a message to the console:

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *message;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit (1);
    }
    message = argv[1];

    if (!printmessage(message)) {
        fprintf(stderr, "%s: couldn't print your message\n",
            argv[0]);
        exit (1);
    }
    printf("Message Delivered!\n");
    exit (0);
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was actually printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL) {
        return (0);
    }
    fprintf(f, "%s\n", msg);
    fclose(f);
    return (1);
}
```

And then, of course:

```
example%  cc printmsg.c -o printmsg
example%  printmsg "Hello, there."
Message delivered!
example%
```

If `printmessage()` was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, one would just like to stick a keyword like `remote` in front of a procedure to turn it into a remote procedure. Unfortunately, we have to live within the constraints of the C language, since it existed long before RPC did. But even without language support, it's not very difficult to make a procedure remote.

In general, it's necessary to figure out what the types are for all procedure inputs and outputs. In this case, we have a procedure `printmessage()` which takes a string as input, and returns an integer as output. Knowing this, we can write a protocol specification in RPC language that describes the remote version of `printmessage()`. Here it is:

```
/*
 * msg.x: Remote message printing protocol
 */

program MESSAGEPROG {
    version MESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 0x20000099;
```

Remote procedures are part of remote programs, so we actually declared an entire remote program here which contains the single procedure `PRINTMESSAGE`. By convention, all RPC services provide for procedure 0. It is normally used for pinging purposes. The above procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary in the protocol definition because `rpcgen` generates it automatically and the user is not concerned with it.

Notice that everything is declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is "string" and not "char *". This is because a "char *" in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a "string".

There are just two more things to write. First, there is the remote procedure itself. Here's the definition of a remote procedure to implement the `PRINTMESSAGE` procedure we declared above.

```
/*
 * msg_proc.c: implementation of the remote procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>      /* always needed   */
#include "msg.h"          /* msg.h will be generated by rpcgen  */

/*
 * Remote verson of "printmessage"
 */
int *
printmessage_1(msg)
     char **msg;
{
     static int result; /* must be static! */
     FILE *f;

     f = fopen("/dev/console", "w");
     if (f == NULL) {
          result = 0;
          return (&result);
     }
     fprintf(f, "%s\n", *msg);
     fclose(f);
     result = 1;
     return (&result);
}
```

Notice here that the declaration of the remote procedure `printmessage_1()` differs from that of the local procedure `printmessage()` in three ways:

1. It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves. If there are no arguments, specify `void`.
2. It returns a pointer to an integer instead of an integer itself. This is also characteristic of remote procedures — they return pointers to their results. Therefore it is important to have the result declared as a `static`. If there are no arguments, specify `void`.
3. It has an "_1" appended to its name. In general, all remote procedures called by `rpcgen` are named by the following rule: the name in the procedure definition (here `PRINTMESSAGE`) is converted to all lower-case letters, an underbar ("_") is appended to it, and finally the version number (here 1) is appended.

The last thing to do is declare the main client program that will call the remote procedure. Here it is:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
```

```
#include <stdio.h>
#include <rpc/rpc.h>        /* always needed */
#include "msg.h"            /* msg.h will be generated by rpcgen */

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

    if (argc != 3) {
        fprintf(stderr,
        "usage: %s host message\n", argv[0]);
        exit(1);
    }

    server = argv[1];
    message = argv[2];

    /*
     * Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RCP package
     * to use the "tcp" protocol when contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS,
      "tcp");
    if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*
     * Call the remote procedure "printmessage" on the server
     */
    result = printmessage_1(&message, cl);
    if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }

    /*
     * Okay, we successfully called the remote procedure.
     */
    if (*result == 0) {
        /*
```

```
        * Server was unable to print our message.
        * Print error message and die.
        */
    fprintf(stderr, "%s: %s couldn't print your message\n",
        argv[0], server);
        exit(1);
    }
    /*
     * The message got printed on the server's console
     */
    printf("Message delivered to %s!\n", server);
    exit(0);
}
```

There are a few points worth noting here:

1. First a client "handle" is created using the RPC library routine `clnt_create()`. This client handle will be passed to the skeleton routines which call the remote procedure.
2. The last parameter to `clnt_create` is "tcp", the transport on which you want your application to run on. It could also have been "udp", as an alternate transport. For more information on transport selection see the section *Transport Selection* in Chapter *2 — Introduction to Remote Procedure Calls*.
3. The remote procedure `printmessage_1()` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the second argument.
4. The remote procedure call can fail in two ways. The RPC mechanism itself can fail or, alternatively, there can be an error in the execution of the actual remote procedure. In the former case, the remote procedure (in this case `print_message_1()`) returns with a `NULL`. In the later case, however, the details of error reporting are application dependent. Here, the error is being reported via **result*.

Here's how to put all of the pieces together:

```
example%  rpcgen msg.x
example%  cc rprintmsg.c msg_clnt.c -o rprintmsg
example%  cc msg_proc.c msg_svc.c -o msg_server
```

Two programs were compiled here: the client program `rprintmsg` and the server program `msg_server`. Before doing this though, `rpcgen` was used to fill in the missing pieces.

Here is what `rpcgen` (called without any flags) did with the input file `msg.x`:

1. It created a header file called `msg.h` that contained `#define`'s for `MESSAGEPROG`, `MESSAGEVERS` and `PRINTMESSAGE` for use in the other modules. This file should be included by *both* the client and the server

modules.

2. It created the client "skeleton" routines in the `msg_clnt.c` file. In this case there is only one, the `printmessage_1()` that was referred from the `printmsg` client program. If the name of the input file is `FOO.x`, the client skeletons output file is called `FOO_clnt.c`.
3. It created the server program in `msg_svc.c` which calls `printmessage_1()` from `msg_proc.c`. The rule for naming the server output file is similar to the previous one: for an input file called `FOO.x`, the output server file is named `FOO_svc.c`.

(Note that, given the `-T` argument, `rpcgen` creates an additional output file which contains index information used for the dispatching of service routines).

Now we're ready to have some fun. First, copy the server to a remote machine and run it. For this example, the machine is called "moon".

Note that servers generated by `rpcgen` can be invoked with port-monitors like `inetd` as well as from the command line, if they are invoked with the `-I` option.

```
moon% msg_server &
```

Then on our local machine ("sun") we can print a message on "moon"'s console.

```
sun% rprintmsg moon "Hello, moon."
```

The message will get printed on "moon"'s console. You can print a message on anybody's console (including your own) with this program if you can copy the server to their machine and run it.

An Advanced Example

The previous example only demonstrated the automatic generation of client and server RPC code. `rpcgen` may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This next example is more advanced in that it presents a complete RPC service—a remote directory listing service, which uses `rpcgen` not only to generate skeleton routines, but also to generate the XDR routines. Here is the protocol description file.

```
/*
 * dir.x: Remote directory listing protocol
 */
const MAXNAMELEN = 255;        /* maximum length of a directory entry */

typedef string nametype<MAXNAMELEN>;      /* a directory entry */

typedef struct namenode *namelist;        /* a link in the listing */

/*
 * A node in the directory listing
 */
struct namenode {
    nametype name;        /* name of directory entry */
    namelist next;        /* next entry */
};
```

```
/*
 * The result of a READDIR operation.
 */
union readdir_res switch (int errno) {
case 0:
    namelist list;   /* no error: return directory listing */
default:
    void;            /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
    version DIRVERS {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 0x20000076;
```

NOTE *Types (like* `readdir_res` *in the example above) can be defined using the "struct", "union" and "enum" keywords, but those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union "foo", you should declare using only "foo" and not "union foo". In fact,* `rpcgen` *compiles RPC unions into C structures and it is an error to declare them using the "union" keyword.*

Running `rpcgen` on `dir.x` creates four output files. First are the basic three itemized above: those containing the header file, client skeleton routines and server skeleton. The fourth contains the XDR routines necessary for converting the data types we declared into XDR format and vice-versa. These are output in the file `dir_xdr.c`. For each data type used in the `.x` file, `rpcgen` assumes that the RPC/XDR library has a routine defined with the name of that data type prepended by `xdr_` (e.g. `xdr_int`). If the data type was defined in the `.x` file, then `rpcgen` will generate the required xdr routine. If there are no such data types, then the file (e.g. `dir_xdr.c`) will not be generated. If the data types were used but not defined, then the user has to provide that xdr routine. This is a way for users to provide their own customized xdr routines.

Here is the implementation of the `READDIR` procedure.

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>          /* Always needed */
#include <sys/dir.h>
#include "dir.h"              /* Created by rpcgen */

extern int errno;
extern char *malloc();
extern char *strdup();
```

```
readdir_res *
readdir_1(dirname)
    nametype *dirname;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

    /*
     * Open directory
     */
    dirp = opendir(*dirname);
    if (dirp == NULL) {
        res.errno = errno;
        return (&res);
    }

    /*
     * Free previous result
     */
    xdr_free(xdr_readdir_res, &res);

    /*
     * Collect directory entries.
     * Memory allocated here will be freed by xdr_free
     * next time readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }
    *nlp = NULL;

    /*
     * Return the result
     */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

Finally, there is the client side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h>      /* always need this */
#include "dir.h"          /* will be generated by rpcgen */
```

```
extern int errno;

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n",
          argv[0]);
        exit(1);
    }

    server = argv[1];
    dir = argv[2];

    /*
     * Create client "handle" used for calling DIRPROG on the
     * server designated on the command line. Use the tcp protocol when
     * contacting the server.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*
     * Call the remote procedure readdir on the server
     */
    result = readdir_1(&dir, cl);
    if (result == NULL) {
        /*
         * An RPC error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }

    /*
     * Okay, we successfully called the remote procedure.
     */
    if (result->errno != 0) {
        /*
         * A remote system error occurred.
         * Print error message and die.
```

```
        */
        errno = result->errno;
        perror(dir);
        exit(1);
    }
    /*
     * Successfully got a directory listing.
     * Print it out.
     */
    for (nl = result->readdir_res_u.list; nl != NULL;
      nl = nl->next) {
        printf("%s\n", nl->name);
    }
    exit(0);
}
```

Compile everything, and run.

```
sun%  rpcgen dir.x
sun%  cc -c dir_xdr.c
sun%  cc rls.c dir_clnt.c dir_xdr.o -o rls
sun%  cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc
sun%  dir_svc &
moon%  rls sun /usr/pub
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

`rpcgen` generated client code does not release the memory allocated for the results of the RPC call. Users can call `xdr_free` to free up the memory once they are done with it. It is quite similar to calling `free()` except that here one also has to pass the xdr routine for the result. In this example, after printing the list, the user could have called

```
xdr_free(xdr_readdir_res, result);
```

Debugging Applications

It is often difficult to debug distributed applications like these because the client and the server are two different processes. To simplify the testing and debugging process, the client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server skeletons. This could be done in the previous example by doing:

```
cc rls.c dir_clnt.c dir_proc.c dir_xdr.c -o rls
```

The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger such as `dbxtool`. When the program is working, the client program can be linked to the client skeleton produced by `rpcgen` and the server procedures can be linked to the server skeleton produced by `rpcgen`.

NOTE *If you do this, you will have to comment out calls to client create RPC library routines (e.g.* `clnt_create()`*).*

There are two kinds of errors which can happen in an RPC call. The first kind of error is caused if there is some problem with the actual mechanism of the remote procedure calls. This could happen in such cases as the procedure is not available, the remote server is not responding, the remote server is unable to decode the arguments, and so on. In the previous example, an RPC error has occurred if `result` is `NULL`. The reason for the failure can be printed by using `clnt_perror()`, or an error string can be returned through `clnt_sperror()`.

The second type of error is due to the server itself. In the previous example, an error was reported if `opendir()` fails. Now you can see why `readdir_res` is of type `union`. The handling of these types of errors are the responsibility of the programmer.

The C-Preprocessor

The C-preprocessor, `cpp`, is run on all input files before they are compiled, so all the preprocessor directives are legal within a ".x" file. Five macro identifiers may have been defined, depending upon which output file is getting generated. They are:

Identifier	*Usage*
`RPC_HDR`	For header-file output
`RPC_XDR`	For XDR routine output
`RPC_SVC`	For server-skeleton output
`RPC_CLNT`	For client skeleton output
`RPC_TBL`	For index table output

Also, `rpcgen` does some additional preprocessing of the input file. Any line that begins with a percent sign is passed directly into the output file, without any interpretation of the line. Here is a simple example that demonstrates this processing feature.

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%       static int thetime;
%
%       thetime = time(0);
%       return (&thetime);
%}
#endif
```

When using the '%' feature, there is no guarantee that `rpcgen` will place the output where you intended. If you have problems of this type, we recommend you to not use this feature.

rpcgen Programming Notes

Network Types

By default `rpcgen` generates server code for both UDP & TCP transports. The `-s` flag creates a server which responds to requests on the specified transport. The following example creates a udp server:

```
example#  rpcgen -s udp_n proto.x
```

User-Provided Define Statements

`rpcgen` also provides a means of defining symbols and assigning values to them. These defined symbols are passed on to the C preprocessor when it is invoked. This facility is useful when the user wants to, for example, invoke debugging code which is enabled only when the `DEBUG` symbol is defined. For example:

```
example%  rpcgen -DDEBUG proto.x
```

Inetd Support

`rpcgen` can also be used to create RPC servers which can be invoked by `inetd` when a request for that service comes in.

```
example%  rpcgen -I proto.x
```

The server code in `proto_svc.c` has the required support for `inetd`. For more information on how to setup the entry for RPC services in `/etc/inetd.conf`, please see the *Using Inetd* section of *Remote Procedure Call Programming Guide.*

In many applications, it is useful for services to wait after satisfying a servicing request, on the chance that another will follow. However, if there is no call within the specified time, the server will exit and the portmonitor will continue to monitor requests for its services. By default, services wait of 120 seconds after servicing a request before exiting. The user can, however, change that interval with the `-K` flag.

```
example%  rpcgen -I -K 20 proto.x
```

Here the server will wait only for 20 seconds before exiting. If you want the server to exit immediately, `-K 0` can be used, while if the server is intended to stay around forever (a normal server) the appropriate argument is `-K -1`.

Dispatch Tables

There are a number of cases when dispatch tables are useful. For example, the server dispatch routine may need to check authorization and then invoke the service routine; or a client library may want to deal with the details of storage management and XDR data conversion.

```
example%  rpcgen -T proto.x
```

Here `rpcgen` generates RPC dispatch tables for each program defined in the protocol description file, `proto.x`, in the file `proto_tbl.i`. (The suffix `.i` stands for "index"). See below for how to use this file when compiling programs. Each entry in the table is a `struct rpcgen_table`, defined in the header file `proto.h` as follows:

```
struct rpcgen_table {
    char        *(*proc)();
    xdrproc_t   xdr_arg;
    unsigned    len_arg;
    xdrproc_t   xdr_res;
    unsigned    len_res;
};
```

where

`proc` is a pointer to the service routine,

`xdr_arg` is a pointer to the input (argument) xdr_routine,

`len_arg` is the length in bytes of the input argument,

`xdr_res` is a pointer to the output (result) xdr_routine, and

`len_res` is the length in bytes of the output result.

The table, named `dirprog_1_table`, is indexed by procedure number. The variable `dirprog_1_nproc` contains the number of entries in the table.

An example of how to locate an procedure in the dispatch tables is demonstrated by the routine `find_proc`:

```
struct rpcgen_table *
find_proc(proc)
    long    proc;
{
    if (proc >= dirprog_1_nproc)
        /* error */
    else
        return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, the service routine is not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the actual service routine initializer is `RPCGEN_ACTION(proc_ver)`.

This way, the same dispatch table can be included in both the client and the server. Use the following define when compiling the client:

```
#define RPCGEN_ACTION(routine)  0
```

and use this define when compiling the server:

```
#define RPCGEN_ACTION(routine)  routine
```

Client Programming Notes

Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when `clnt_create()` is used. This means RPC will wait for 25 seconds to get the results from the server. If it does not hear within that time period, then perhaps the server isn't running or the remote machine crashed or the network is unreachable. There are many possibilities of why no answer is heard. In such cases the function will return NULL and the error can be printed using `clnt_perrno()`.

There are cases when the user wants to change the timeout value to accommodate the application needs or the fact that the server is slow and quite far away. The

timeout can be changed using `clnt_control()`. Here is a small code fragment to demonstrate use of `clnt_control()`:

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL) {
    exit(1);
}
tv.tv_sec = 60; /* change timeout to 1 minute */
tv.tv_usec = 0; /* this should always be set */
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Client Authentication

The client create routines do not, by default, have any facilities for client authentication, but the client may sometimes want to authenticate itself to the server. For more information on how to perform authentication, see the *Authentication* section of *Remote Procedure Call Programming Guide.* Doing so is trivial, and looks like this:

```
CLIENT *cl;

cl = client_create("somehost", SOMEPROG, SOMEVERS, "udp");
if (cl != NULL) {
    /* To set UNIX style authentication */
    cl->cl_auth = authunix_create_default();
}
```

Server Programming Notes

Handling Broadcast on the Server Side

Clients may sometimes broadcast to find out whether a particular server exists on the network or just to find out about all the servers for a particular program and version number. These calls are made via `clnt_broadcast()`. Note that there is no rpcgen support for that. Please see *Broadcast RPC Synopsis* in *Remote Procedure Call Programming Guide.*

When a procedure is known to be called via broadcast RPC, it is usually wise for the server to not reply unless it can provide some useful information to the client. This prevents the network from getting flooded by useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by `rpcgen` will detect this and not send out a reply.

Here is an example of a procedure that replies only if it thinks it is an NFS server:

```
void *
reply_if_nfsserver()
{
    char notnull;    /* just here so we can use its address */

    if (access("/etc/exports", F_OK) < 0) {
        return (NULL);  /* prevent RPC from replying */
    }
    /*
     * return non-null pointer so RPC will send out a reply
     */
    return ((void *)&notnull);
}
```

Note that if procedure returns type "void *", they must return a non-NULL pointer if they want RPC to reply for them.

Other Information Passed to Server Procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security. This extra information is actually supplied to the server procedure as a second argument. (For details see the structure of `svc_req`, in the *Authentication* section of *Remote Procedure Call Programming Guide*. Here is an example to demonstrate its use. What we've done here is rewrite the previous `printmessage_1()` procedure to only allow root users to print a message to the console.

```
int *
printmessage_1(msg, rqstp)
    char **msg;
    struct svc_req  *rqstp;
{
    static int result;  /* Must be static */
    FILE *f;
    struct authunix_parms *aup;

    aup = (struct authunix_parms *)rqstp->rq_clntcred;
    if (aup->aup_uid != 0) {
        result = 0;
        return (&result);
    }

    /*
     * Same code as before.
     */
}
```

RPC Language

RPC language is an extension of XDR language. The sole extension is the addition of the `program` and `version` types. For a complete description of the XDR language syntax, see the *External Data Representation Standard: Protocol Specification* chapter. For a description of the RPC extensions to the XDR language, see the *Remote Procedure Calls: Protocol Specification* chapter.

However, XDR language is so close to C that if you know C, you know most of it already. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

Definitions

An RPC language file consists of a series of definitions.

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes the following types of definitions.

```
definition:
    enum-definition
    typedef-definition
    const-definition
    declaration-definition
    struct-definition
    union-definition
    program-definition
```

Enumerations

XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into.

```
 enum colortype {          enum colortype {
      RED = 0,                  RED = 0,
      GREEN = 1,    -->         GREEN = 1,
      BLUE = 2                  BLUE = 2,
 };                        };
                           typedef enum colortype colortype;
```

Typedefs

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
    "typedef" declaration
```

Here is an example that defines a `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Constants

XDR constants may be used wherever a integer constant is used, for example, in array size specifications.

```
const-definition:
    "const" const-ident "=" integer
```

For example, the following defines a constant `DOZEN` equal to 12.

```
const DOZEN = 12;  -->  #define DOZEN 12
```

Declarations

In XDR, there are only four kinds of declarations.

```
declaration:
    simple-declaration
    fixed-array-declaration
    variable-array-declaration
    pointer-declaration
```

1) Simple declarations are just like simple C declarations.

```
simple-declaration:
    type-ident variable-ident
```

Example:

```
colortype color;    --> colortype color;
```

2) Fixed-length Array Declarations are just like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];   --> colortype palette[8];
```

3) Variable-Length Array Declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;     /* at most 12 items */
int widths<>;        /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are

actually compiled into "struct"s. For example, the "heights" declaration gets compiled into the following struct:

```
struct {
    u_int heights_len;   /* # of items in array */
    int *heights_val;    /* pointer to array */
} heights;
```

Note that the number of items in the array is stored in the "_len" component and the pointer to the array is stored in the "_val" component. The first part of each of these component's names is the same as the name of the declared XDR variable.

4) **Pointer Declarations** are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called "optional-data", not "pointer", in XDR language.

```
pointer-declaration:
    type-ident "*" variable-ident
```

Example:

```
listitem *next;  -->  listitem *next;
```

Structures

An XDR struct is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

As an example, here is an XDR structure to a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {              struct coord {
    int x;        -->           int x;
    int y;                      int y;
};                          };
                            typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This allows one to use "coord" instead of "struct coord" in declarations.

Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
    "union" union-ident "switch" "(" simple declaration ")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "case" value ":" declaration ";" case-list
    "default" ":" declaration ";"
```

Here is an example of a type that might be returned as the result of a "read data" operation. If there is no error, return a block of data. Otherwise, don't return anything.

```
union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

It gets compiled into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the structure type name, except for the trailing "_u".

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value
```

For example, here is the time protocol, revisited:

```
/*
 * time.x: Get or set the time. Time is represented as number of seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into these #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Special Cases

There are a few exceptions to the rules described above.

Booleans: C has no built-in boolean type. However, the RPC library has a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Things declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married;  -->  bool_t married;
```

Strings: C has no built-in string type, but instead uses the null-terminated "char *" convention. In XDR language, strings are declared using the "string" keyword, and compiled into "char *"s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;  -->  char *name;
string longname<>;  -->  char *longname;
```

Opaque Data: Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.

```
Examples:
    opaque diskblock[512];  -->  char diskblock[512];

    opaque filedata<1024>;  -->  struct {
                                     u_int filedata_len;
                                     char *filedata_val;
                                 } filedata;
```

Voids: In a void declaration, the variable is not named. The declaration is just "void" and nothing else. Void declarations can only occur in two places: union

definitions and program definitions (as the argument or result of a remote procedure).

4

Remote Procedure Call Programming Guide

This document assumes a working knowledge of network theory. It is intended for programmers who wish to write network applications using remote procedure calls (explained below), and who want to understand the RPC mechanisms usually hidden by the `rpcgen(1)` protocol compiler. `rpcgen` is described in detail in the previous chapter, the **`rpcgen`** *Programming Guide.*

NOTE *Before attempting to write a network application, or to convert an existing non-network application to run over the network, you may want to understand the material in this chapter. However, for most applications, you can circumvent the need to cope with the details presented here by using* `rpcgen`. *The An Advanced Example section of that chapter contains the complete source for a working RPC service—a remote directory listing service which uses* `rpcgen` *to generate XDR routines as well as client and server stubs.*

4.1. Layers of RPC

The RPC interface can be seen as being divided into three layers.[6]

The Highest Layer: The highest layer is totally transparent to the operating system, machine and network upon which is run. It's probably best to think of this level as a way of *using* RPC, rather than as a *part of* RPC proper. Programmers who write RPC routines should (almost) always make this layer available to others by way of a simple C front end that entirely hides the networking.

To illustrate, at this level a program can simply make a call to `rnusers()`, a C routine which returns the number of users on a remote machine. The user is not explicitly aware of using RPC — they simply call a procedure, just as they would call `malloc()`.

The Middle Layer: The middle simplified layer is really "RPC proper." Here, the user doesn't need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The selling point here is simplicity. It's this layer that allows RPC to pass the "hello world" test — simple things should be simple. The middle layer routines are used for most applications.

Simplified RPC calls are made with the system routines `registerrpc()`, `callrpc()` and `svc_run()`. `registerrpc()` obtains a unique system-

[6] For a complete specification of the routines in the remote procedure call Library, see the `rpc(3N)` manual page.

wide procedure-identification number, and `callrpc()` actually executes a remote procedure call. At the middle level, a call to `rnusers()` is implemented by way of these two routines.

The middle layer is rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It doesn't support multiple kinds of call authentication. The programmer rarely needs all these kinds of control, but one or two of them is often necessary.

The Lowest Layer: The lowest layer does allow these details to be controlled by the programmer. Programs written at this level are also most efficient and allow for flexibility. The lowest layer routines include client creation routines such as `clnt_create()`, the actual client call `clnt_call()`, server creation routines such as `svcudp_create()`, and the server registration routine `svc_register()`.

Higher Layers of RPC

This layer consists of RPC-library based services. Imagine you're writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the RPC library routine `rnusers()`, as illustrated below:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

    if (argc != 2) {
        fprintf(stderr, "usage: rnusers hostname\n");
        exit(1);
    }
    if ((num = rnusers(argv[1])) < 0) {
        fprintf(stderr, "error: rnusers\n");
        exit(1);
    }
    printf("%d users on %s\n", num, argv[1]);
    exit(0);
}
```

RPC library routines such as `rnusers()` are in the RPC services library `librpcsvc.a`. Thus, the program above should be compiled with

```
example% cc program.c -lrpcsvc
```

`rnusers()`, like the other RPC library routines, is documented in section 3R of the *System Services Overview*, the same section which documents the standard Sun RPC services. See the `intro(3R)` manual page for an explanation of the documentation strategy for these services and their RPC protocols.

Here are some of the RPC service library routines available to the C programmer:

Table 4-1 *RPC Service Library Routines*

Routine	*Description*
`rnusers`	Return number of users on remote machine
`rusers`	Return information about users on remote machine
`havedisk`	Determine if remote machine has disk
`rstat`	Get performance data from remote kernel
`rwall`	Write to specified remote machines
`yppasswd`	Update user password in Network Information Service

Other RPC services — for example `ether`, `mount`, `rquota`, and `spray` — are not available to the C programmer as library routines. They do, however, have RPC program numbers so they can be invoked with `callrpc()`, which will be discussed in the next section. Most of them also have compilable `rpcgen(1)` protocol description files. Some of the files (in the form *.x) may be found in `/usr/include/rpcsvc`. (The `rpcgen` protocol compiler radically simplifies the process of developing network applications. See the **`rpcgen`** *Programming Guide* chapter for detailed information about `rpcgen` and `rpcgen` protocol description files).

Middle Layers of RPC

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc()` and `registerrpc()`. Using this method, the number of remote users can be obtained as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;
    int stat;

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(1);
    }
    if (stat = callrpc(argv[1],
      RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
      xdr_void, 0, xdr_u_long, &nusers) != 0) {
        clnt_perrno(stat);
        exit(1);
    }
    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

The simplest way of making remote procedure calls is with the RPC library routine `callrpc()`. It has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers—together they identify the procedure to be called. The fifth and sixth parameters are an XDR filter and an argument to be encoded and passed to the remote procedure. XDR filter is a user provided procedure which can encode or decode machine native data to or from the XDR format. The final two parameters are an XDR filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If `callrpc()` completes successfully, it returns zero; else it returns a nonzero value. The return codes are found in `<rpc/clnt.h>`.

`callrpc()` needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For `RUSERSPROC_NUM`, the return value is an `unsigned long`, so `callrpc()` has `xdr_u_long()` as its first return parameter, which says that the result is of type `unsigned long`, and `&nusers` as its second return parameter, which is a pointer to where the long result will be placed. Since `RUSERSPROC_NUM` takes no argument, the argument parameter of `callrpc()` is `xdr_void`. In such cases the argument should be NULL.

After trying several times to deliver a message, if `callrpc()` gets no answer, it returns with an error code. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document.

The remote server procedure corresponding to the above might look like this:

```
unsigned long *
nuser(indata)
     char *indata;
{
     static unsigned long nusers;

     /*
      * Code here to compute the number of users
      * and place result in variable nusers.
      */
     return(&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in our example), and it returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so input argument and the return value can be cast to `char *`.

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. If `rpcgen` is used to provide this functionality, it will also generate a server dispatch function. But users can write the servers themselves using `registerrpc()` and especially so for

simple applications like the one shown here. In this example, there is only a single procedure to register, so the main body of the server would look like this:

```
#include <stdio.h>
#include <rpc/rpc.h>          /* required */
#include <rpcsvc/rusers.h>    /* for prog, vers definitions */

unsigned long *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
        nuser, xdr_void, xdr_u_long);
    svc_run();        /* Never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

The `registerrpc()` routine registers a procedure as corresponding to a given RPC procedure number. The first three parameters, `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM` are the program, version, and procedure numbers of the remote procedure to be registered; `nuser()` is the name of the local procedure that implements the remote procedure; and `xdr_void()` and `xdr_u_long()` are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures).

The underlying transport mechanism used with `registerrpc()` is both `callrpc()` and UDP.

WARNING ***Warning: the UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.***

After registering the local procedure, the server program's main procedure calls `svc_run()`, the RPC library's remote procedure dispatcher. It is this function that calls the remote procedures in response to RPC requests. Note that the dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered with `registerrpc()`.

Passing Arbitrary Data Types

In the previous example, the RPC passes a single `unsigned long`. RPC can handle arbitrary data structures, regardless of different machine's byte orders or structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in the previous example, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()       xdr_u_int()       xdr_enum()
xdr_long()      xdr_u_long()      xdr_bool()
xdr_short()     xdr_u_short()     xdr_wrapstring()
xdr_char()      xdr_u_char()
```

Note that the routine `xdr_string()` exists, but cannot be used with `callrpc()` and `registerrpc()`, which only pass two parameters to their XDR routines. Instead `xdr_wrapstring()` can be used. It takes only two parameters, and is thus OK. It calls `xdr_string()`.

As an example of a user-defined type routine, if you wanted to send the structure

```
struct simple {
    int a;
    short b;
} simple;
```

then you would call `callrpc()` as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_simple, &simple ...);
```

where `xdr_simple()` is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (0);
    if (!xdr_short(xdrsp, &simplep->b))
        return (0);
    return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in the *XDR Protocol Specification* section of this manual, only few implementation examples are given here.

NOTE *We strongly recommend that* `rpcgen` *be used to generate XDR routines. The "-c" option of* `rpcgen` *can be used to generate just the* `_xdr.c` *file.*

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()        xdr_bytes()      xdr_reference()
xdr_vector()       xdr_union()      xdr_pointer()
xdr_string()       xdr_opaque()
```

To send a variable array of integers, you might package them up as a structure like this

```
struct varintarr {
    int *data;
    int arrlnth;
} arr;
```

and make an RPC call such as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_varintarr, &arr...);
```

with `xdr_varintarr()` defined as:

```
xdr_varintarr(xdrsp, arrp)
    XDR *xdrsp;
    struct varintarr *arrp;
{
    return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
        MAXLEN, sizeof(int), xdr_int));
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, one can use `xdr_vector()`, which serializes fixed-length arrays.

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
    XDR *xdrsp;
    int intarr[];
{
    return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
        xdr_int));
}
```

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine `xdr_bytes()`, which is like `xdr_array()` except that it packs characters;

`xdr_bytes()` has four parameters, similar to the first four parameters of `xdr_array()`. For null-terminated strings, there is also the `xdr_string()` routine, which is the same as `xdr_bytes()` without the length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written `xdr_simple()` as well as the built-in functions `xdr_string()` and `xdr_reference()`, which chases pointers:

```
struct finalexample {
    char *string;
    struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{

    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (0);
    if (!xdr_reference(xdrsp, &finalp->simplep,
      sizeof(struct simple), xdr_simple);
        return (0);
    return (1);
}
```

Note that we could as easily call `xdr_simple()` here instead of `xdr_reference()`.

Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library.

There are several occasions when you may need to use lower layers of RPC. First, you may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data. For an example, see the *TCP* section below. Second, you may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see the *Memory Allocation with XDR* section below. Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in the *Authentication* section below.

More on the Server Side

The server for the `nusers()` program shown below does the same thing as the one using `registerrpc()` above, but is written using a lower layer of the RPC package:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main()
{
    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
            nuser, IPPROTO_UDP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    case RUSERSPROC_NUM:
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. If the argument to `svcudp_create()` is `RPC_ANYSOCK`, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` and `clntudp_create()` (the low-level client routine) must match. `registerrpc()` uses `svcudp_create()` to get a UDP handle. If you require a more reliable protocol, call `svctcp_create()` instead.

After creating an `SVCXPRT`, the next step is to call `pmap_unset()` so that if the `nusers()` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset()` erases the entry for `RUSERSPROG` from the portmapper's tables.

Finally, we associate the program number `RUSERSPROG` and version `RUSERSVERS` with the procedure `nuser()`, which in this case, is `IPPROTO_UDP`. Notice that unlike `registerrpc()`, there are no XDR routines involved in the registration process. Also, registration is done on the program level rather than procedure level. A service may choose to register its port number with the local portmapper service. This is done by specifying a non-zero protocol number in the final argument of `svc_register()`. A client can discover the server's port number by consulting the portmapper on their server's machine. This can be done automatically by specifying a zero port number in `clntudp_create()` or `clnttcp_create()`.

The user routine `nuser()` must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by `nuser()` that `registerrpc()` handles automatically. The first is that procedure `NULLPROC` (currently zero) returns with no results. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. Its first parameter is the `SVCXPRT` handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Note that it is not required to have `nusers` declared as `static` here because `svc_sendreply()` is called within that function itself. Not illustrated above is how a server handles an RPC program that receives data. As an example, we can add a procedure `RUSERSPROC_BOOL`, which has an argument `nusers()`, and returns `TRUE` or `FALSE` depending on whether there are nusers logged on. It would look like this:

```
case RUSERSPROC_BOOL: {
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
```

```
        svcerr_decode(transp);
        return;
    }
    /*
     * Code to set nusers = number of users
     */
    if (nuserquery == nusers)
        bool = TRUE;
    else
        bool = FALSE;
    if (!svc_sendreply(transp, xdr_bool, &bool))
         fprintf(stderr, "can't reply to RPC call\n");
    return;
}
```

The relevant routine is `svc_getargs()`, which takes an `SVCXPRT` handle, the XDR routine, and a pointer to where the input is to be placed as arguments.

More on the Client Side

When you use `callrpc()`, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider the following code to call the `nusers` service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "can't get addr for %s\n",argv[1]);
        exit(-1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
```

```
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
      RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror("clntudp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    printf("%d users on %s\n", nusers, argv[1]);
    clnt_destroy(client);
    exit(0);
}
```

The `CLIENT` pointer is encoded with the transport mechanism. `callrpc()` uses UDP, thus it calls `clntudp_create()` to get a `CLIENT` pointer. To get TCP you would use `clnttcp_create()`.

The parameters to `clntudp_create()` are the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. Only when the `sin_port` is 0, the remote portmapper is queried to find out the address of the remote service.

The low-level version of `callrpc()` is `clnt_call()`, which takes a `CLIENT` pointer rather than a host name. The parameters to `clnt_call()` are a `CLIENT` pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply. If the client does not hear from the server within the time specified in `pertry_timeout`, the request may be sent again to the server. Thus, the number of tries that the `clnt_call()` will make to contact the server is the `clnt_call()` timeout divided by the `clntudp_create()` timeout.

Note that the `clnt_destroy()` call always deallocates the space associated with the `CLIENT` handle. It closes the socket associated with the `CLIENT` handle only if the RPC library opened it. It the socket was opened by the user, it stays open. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, the call to `clntudp_create()` is replaced with `clnttcp_create()`.

```
clnttcp_create(&server_addr, prognum, versnum, &sock,
                inbufsize, outbufsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that `CLIENT` handle would use this connection. The server side of an RPC call using TCP has `svcudp_create()` replaced by `svctcp_create()`.

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to `svctcp_create()` are send and receive sizes respectively. If '0' is specified for either of these, the system chooses default values.

The simplest routine to create a client handle is `clnt_create()`.

```
clnt=clnt_create(server_host, prognum, versnum, transport);
```

The parameters are the name of the host on which the service resides, the program and version number and the transport to be used. The transport can be either "udp" for UDP or "tcp" for TCP. It is possible to change the default timeouts using `clnt_control()`. For more details look under *Client Programming Notes* section in **rpcgen** *Programming Guide*.

Memory Allocation with XDR

XDR routines not only do input and output, they may also do memory allocation. This is why the second parameter of `xdr_array()` is a pointer to an array, rather than the array itself. If it is `NULL`, then `xdr_array()` allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine `xdr_chararr1()`, which deals with a fixed array of bytes with length `SIZE`:

```
xdr_chararr1(xdrsp, chararr)
    XDR *xdrsp;
    char chararr[];
{
    char *p;
    int len;

    p = chararr;
    len = SIZE;
    return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in *chararr*, it can be called from a server like this:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

If you want XDR to do the allocation, you would have to rewrite this routine in the following way:

```
xdr_chararr2(xdrsp, chararrp)
    XDR *xdrsp;
    char **chararrp;
{
    int len;

    len = SIZE;
    return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * Use the result here
 */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

Note that, after being used, the character array can be freed with `svc_freeargs()`. `svc_freeargs()` will not attempt to free any memory if the variable indicating it is NULL. For example, in the the routine `xdr_finalexample()`, given earlier, if `finalp->string` was NULL, then it would not be freed. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from `callrpc()`, the serializing part is used. When called from `svc_getargs()`, the deserializer is used. And when called from `svc_freeargs()`, the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. See the *External Data Representation: Sun Technical Notes* chapter for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behavior accordingly.

4.2. Raw RPC

Finally, there are two pseudo-RPC interface routines which are intended only for testing purposes. These routines, `clntraw_create()` and `svcraw_create()`, don't actually involve the use of any real transport at all. They exist to help the developer debug and test the non-communications oriented aspects of their application before running it over a real network. Here's an example of their use:

```
/*
 * A simple program to increment the number by 1
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>     /* required for raw */

struct timeval TIMEOUT = {0, 0};
static void server();

main()
{
    CLIENT *clnt;
    SVCXPRT *svc;
    int num = 0, ans;

    if (argc == 2)
        num = atoi(argv[1]);
    svc = svcraw_create();
    if (svc == NULL) {
        fprintf(stderr, "Couldnot create server handle\n");
        exit(1);
    }
    svc_register(svc, 200000, 1, server, 0);
    clnt = clntraw_create(200000, 1);
    if (clnt == NULL) {
        clnt_pcreateerror("raw");
        exit(1);
    }
    if (clnt_call(clnt, 1, xdr_int, &num, xdr_int, &num1,
      TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt, "raw");
        exit(1);
    }
    printf("Client: number returned %d\n", num1);
    exit(0) ;
}

static void
server(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    int num;

    switch(rqstp->rq_proc) {
    case 0:
        if (svc_sendreply(transp, xdr_void, 0) == NULL) {
            fprintf(stderr, "error in null proc\n");
            exit(1);
        }
        return;
    case 1:
```

```
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
    if (!svc_getargs(transp, xdr_int, &num)) {
        svcerr_decode(transp);
        return;
    }
    num++;
    if (svc_sendreply(transp, xdr_int, &num) == NULL) {
        fprintf(stderr, "error in sending answer\n");
        exit(1);
    }
    return;
}
```

Note the following points:

1. All the RPC calls occur within the same thread of control.
2. `svc_run()` is not called.
3. It is necessary that the server be created before the client.
4. `svcraw_create()` takes no parameters.
5. The last parameter to `svc_register` is 0, which means that it will not register with `portmapper`.
6. The server dispatch routine is the same as it is for normal RPC servers.

4.3. Other RPC Features

This section discusses some other aspects of RPC that are useful for the RPC programmer.

Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. But if the other activity involves waiting on a a file descriptor, the `svc_run()` call won't work. The code for `svc_run()` is as follows:

```
void
svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

    for (;;) {
        readfds = svc_fds;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {

        case -1:
            if (errno != EBADF)
```

```
                continue;
            perror("select");
            return;
        case 0:
            continue;
        default:
            svc_getreqset(&readfds);
        }
    }
}
```

You can bypass `svc_run()` and call `svc_getreqset()` yourself. All you need to know are the file descriptors of the socket(s) associated with the programs you are waiting on. Thus you can have your own `select()` that waits on both the RPC socket, and your own descriptors. Note that `svc_fds` is a bit mask of all the file descriptors that RPC is using for services. It can change everytime that *any* RPC library routine is called, because descriptors are constantly being opened and closed, for example for TCP connections.

Caution: if you are handling signals in your application, then either make sure that you do not make any system calls and inadvertently set `errno` or reset `errno` to its old value before returning from your signal handler.

Broadcast RPC

The *portmapper* is a daemon that converts RPC program numbers into DARPA protocol port numbers; see *The Portmapper* section in the *Network Services* chapter. You can't do broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC:

1. Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding server).
2. Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.
5. Broadcast requests are limited in size to 1400 bytes. Replies can be up to 8800 bytes (the current maximum UDP packet size).

Broadcast RPC Synopsis

```
#include <rpc/pmap_clnt.h>
    . . .
enum clnt_stat  clnt_stat;
    . . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
    u_long     prognum;          /* program number */
    u_long     versnum;          /* version number */
    u_long     procnum;          /* procedure number */
    xdrproc_t  inproc;           /* xdr routine for args */
    caddr_t    in;               /* pointer to args */
    xdrproc_t  outproc;          /* xdr routine for results */
    caddr_t    out;              /* pointer to results */
    bool_t     (*eachresult)();  /* call with each result gotten */
```

The procedure `eachresult()` is called each time a response is obtained. It returns a boolean that indicates whether or not the user wants more responses.

```
bool_t done;
    . . .
done = eachresult(resultsp, raddr)
    caddr_t resultsp;
    struct sockaddr_in *raddr; /* Addr of responding server */
```

If *done* is `TRUE`, then broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back in a default total timeout period, the routine returns with `RPC_TIMEDOUT`. You may also refer to *Handling Broadcast on the Server Side* section in the **`rpcgen`** *Programming Guide* **`chapter.`**

Batching

In normal RPC clients send a call message and wait for the server to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgement for every message sent. Actually calls made by clients are buffered, thus causing no processing on the servers. When the connection is flushed, a normal RPC request is sent. The server processes the request and sends the reply back.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called batching. Batching assumes that:

1. Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.
2. The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one `write()` system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a nonbatched call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <suntool/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
      windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can't register WINDOW service\n");
        exit(1);
    }
    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    case RENDERSTRING:
```

```
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /*
             * Tell caller he screwed up
             */
            svcerr_decode(transp);
            return;
        }
        /*
         * Code here to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL))
            fprintf(stderr, "can't reply to RPC call\n");
        break;
    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /*
             * We are silent in the face of protocol errors
             */
            break;
        }
        /*
         * Code here to render string s, but send no reply!
         */
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
    /*
     * Now free string allocated while decoding arguments
     */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course the service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes:

1. the result's XDR routine must be zero (`NULL`),
2. the RPC call's timeout must be zero. Do not rely on `clnt_control()` to assist in batching.

If a UDP transport is used instead, the client call becomes a message to the server and the RPC mechanism reduces to a message passing system. No batching is possible here.

Here is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string (EOF):

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <suntool/windows.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct timeval total_timeout;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;

    if ((client = clnt_create(argv[1],
      WINDOWPROG, WINDOWVERS, "tcp")) == NULL) {
        perror("clnttcp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 0;    /* set timeout to zero */
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batching rpc");
            exit(-1);
        }
    }

    /* Now flush the pipeline */

    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
        xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "batching rpc");
        exit(-1);
    }
    clnt_destroy(client);
    exit(0);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but throw the lines away. The example was run in the following four configurations:

1. machine to itself, regular RPC — 50 seconds

2. machine to itself, batched RPC — 16 seconds
3. machine to another, regular RPC — 52 seconds
4. machine to another, batched RPC — 10 seconds

Running only `fscanf()` on `/etc/termcap` requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type *none*.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support.

UNIX Authentication

The Client Side

When a caller creates a new RPC client handle as in:

```
clnt = clntudp_create(address, prognum, versnum,
              wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to be

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use *UNIX* style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with *clnt* to carry with it the following authentication credentials structure:

```
/*
 * UNIX style credentials.
 */
struct authunix_parms {
    u_long  aup_time;        /* credentials creation time */
    char    *aup_machname;   /* host name where client is */
    int     aup_uid;         /* client's UNIX effective uid */
    int     aup_gid;         /* client's current group id */
    u_int   aup_len;         /* element length of aup_gids */
    int     *aup_gids;       /* array of groups user is in */
};
```

These fields are set by `authunix_create_default()` by invoking the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

This should be done in all cases, to conserve memory.

The Server Side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
    u_long  rq_prog;             /* service program number */
    u_long  rq_vers;             /* service protocol vers num */
    u_long  rq_proc;             /* desired procedure number */
    struct opaque_auth rq_cred;  /* raw credentials from wire */
    caddr_t rq_clntcred;         /* credentials (read only) */
};
```

The `rq_cred` is mostly opaque, except for one field of interest: the style or flavor of authentication credentials:

```
/*
 * Authentication info. Mostly opaque to the programmer.
 */
struct opaque_auth {
    enum_t  oa_flavor;   /* style of credentials */
    caddr_t oa_base;     /* address of more auth stuff */
    u_int   oa_length;   /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

1. That the request's `rq_cred` is well formed. Thus the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the styles supported by the RPC package.
2. That the request's `rq_clntcred` field is either `NULL` or points to a well formed structure that corresponds to a supported style of authentication credentials. `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is `NULL`, the service implementor may wish to inspect the other (opaque) fields of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

Our remote users service example can be extended so that it computes results for all users except UID 16:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    }
    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred =
            (struct authunix_parms *)rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
```

```
        break;
    case AUTH_NULL:
    default:      /* return weak authentication error */
        svcerr_weakauth(transp);
        return;
    }
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:
        /*
         * make sure caller is allowed to call this proc
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call `svcerr_weakauth()`. And finally, the service protocol itself should return status for access denied; in the case of our example, the protocol does not have such a status, so we call the service primitive `svcerr_systemerr()` instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with *authentication* and not with individual services' *access control*. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

DES Authentication

UNIX authentication can be defeated, which we won't explain here. Therefore DES authentication is recommended for people who want more security than what UNIX authentication offers. The details of the DES authentication protocol are complicated and are not explained here. Please see the *Remote Procedure Calls: Protocol Specification* section for the details.

In order for DES authentication to work, the `keyserv(8c)` daemon must be running on both the server and client machines. The users on these machines need public keys assigned by the network administrator in the `publickey(5)` database. And, they need to have decrypted their secret keys using their login password. This automatically happens when one logs in

using `login(1)`, or can be done manually using `keylogin(1)`. The *Network Services* chapter of *Network Programming* explains more how to setup secure networking.

Client Side

If a client wishes to use DES authentication, it must set its authentication handle appropriately. Here is an example:

```
cl->cl_auth =
    authdes_create(servername, 60, &server_addr, NULL);
```

The first argument is the network name or "netname" of the owner of the server process. Typically, server processes are root processes and their netname can be derived using the following call:

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

Here, *rhostname* is the hostname of the machine the server process is running on. `host2netname()` fills in *servername* to contain this root process's netname. If the server process was run by a regular user, one could use the call `user2netname()` instead. Here is an example for a server process with the same user ID as the client:

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

The last argument to both of these calls, `user2netname()` and `host2netname()`, is the name of the naming domain where the server is located. The `NULL` used here means "use the local domain name."

The second argument to `authdes_create()` is a lifetime for the credential. Here it is set to sixty seconds. What that means is that the credential will expire 60 seconds from now. If a user tries to reuse the credential, the server RPC subsystem will recognize that it has expired and not grant any requests. If the same user tries to reuse the credential within the sixty second lifetime, he will still be rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and will not grant requests to duplicates.

The third argument to `authdes_create()` is the address of the host to synchronize with. In order for DES authentication to work, the server and client must agree upon the time. Here we pass the address of the server itself, so the client and server will both be using the same time: the server's time. The argument can be `NULL`, which means "don't bother synchronizing." You should only do this if you are sure the client and server are already synchronized.

The final argument to `authdes_create()` is the address of a DES encryption key to use for encrypting timestamps and data. If this argument is `NULL`, as it is in this example, a random key will be chosen. The client may find out the encryption key being used by consulting the `ah_key` field of the authentication handle.

Server Side

The server side is a lot simpler than the client side. Here is the previous example rewritten to use `AUTH_DES` instead of `AUTH_UNIX`:

```
#include <sys/time.h>
#include <rpc/auth_des.h>
    . . .
    . . .
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];
    /*
     * we don't care about authentication for null proc
     */

    if (rqstp->rq_proc == NULLPROC) {
        /* same as before */
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_DES:
        des_cred =
            (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name,
            &uid, &gid, &gidlen, gidlist)) {
            fprintf(stderr, "unknown user: %s\n",
                des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
```

```
/*
 * The rest is the same as before
 */
```

Note the use of the routine `netname2user()`, the inverse of `user2netname()`: it takes a network ID and converts to a unix ID. `netname2user()` also supplies the group IDs which we don't use in this example, but which may be useful to other UNIX programs.

Using Inetd

An RPC server can be started from `inetd`. The only difference from the usual code is that the service creation routine should be called in the following form:

```
transp = svcudp_create(0);      /* For UDP */
transp = svctcp_create(0,0,0); /* For listener TCP sockets */
transp = svcfd_create(0,0,0);  /* For connected TCP sockets */
```

since `inetd` passes a socket as file descriptor 0. Also, `svc_register()` should be called as

```
svc_register(transp, PROGNUM, VERSNUM, service, 0);
```

with the final flag as 0, since the program would already be registered with `portmapper` by `inetd`. Remember that if you want to exit from the server process and return control to `inetd`, you need to explicitly exit, since `svc_run()` never returns.

The format of entries in `/etc/inetd.conf` for RPC services is in one of the following two forms:

```
p_name/version dgram  rpc/udp wait/nowait user server args
p_name/version stream rpc/tcp wait/nowait user server args
```

where *p_name* is the symbolic name of the program as it appears in `rpc(5)`, *server* is the program implementing the server, and *program* and *version* are the program and version numbers of the service. For more information, see `inetd.conf(5)`.

If the same program handles multiple versions, then the version number can be a range, as in this example:

```
rstatd/1-2 dgram rpc/udp wait root /usr/etc/rpc.rstatd
```

4.4. More Examples

Versions on Server Side

By convention, the first version number of program `PROG` is `PROGVERS_ORIG` and the most recent version is `PROGVERS`. Suppose there is a new version of the `user` program that returns an `unsigned short` rather than a `long`. If we name this version `RUSERSVERS_SHORT`, then a server that wants to support both versions would do a double register. Note that there is no need to create another server handle for the new version.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register new service\n");
    exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            return;
        }
        return;
    case RUSERSPROC_NUM:
    /*
    * Code here to compute the number of users
    * and assign it to the variable nusers
    */
        nusers2 = nusers;
        switch (rqstp->rq_vers) {
        case RUSERSVERS_ORIG:
            if (!svc_sendreply(transp, xdr_u_long,
            &nusers)) {
              fprintf(stderr,"can't reply to RPC call\n");
            }
            break;
        case RUSERSVERS_SHORT:
            if (!svc_sendreply(transp, xdr_u_short,
            &nusers2)) {
              fprintf(stderr,"can't reply to RPC call\n");
            }
            break;
        }
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Versions on Client Side

Since different machines may run different versions of the RPC servers, the client should be prepared to deal with the world. It is possible to have one server running with the old version of `RUSERSPROG(RUSERSVERS_ORIG)` while another server is running with the newer version `(RUSERSVERS_SHORT)`.

If the version of the server running does not match with the version number in the client create routines, then `clnt_call` fails with `RPCPROGVERSMISMATCH` error. You can find out the version numbers supported by the server and then create a client handle with an appropriate version number. Either the routine below can be used, or `clnt_create_vers()`. See the `rpc(3N)` manual page for more details.

```
main()
{
    enum clnt_stat status;
    u_short num_s;
    u_int num_l;
    struct rpc_err rpcerr;
    int maxvers, minvers;

    clnt = clnt_create(host, RUSERSPROG,
        RUSERSVERS_SHORT, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror("clnt");
        exit(-1);
    }
    to.tv_sec = 10; /* set the time outs */
    to.tv_usec = 0;
    status = clnt_call(clnt, RUSERSPROC_NUM,
        xdr_void, NULL, xdr_u_short, &num_s, to);
    if (status == RPC_SUCCESS) {
        /* We found the latest version number */
        clnt_destroy(clnt);
        printf("num = %d\n",num_s);
        exit(0);
    }
    if (status != RPC_PROGVERSMISMATCH) {
        /* Some other error */
        clnt_perror(clnt, "rusers");
        exit(-1);
    }
    clnt_geterr(clnt, &rpcerr);
    maxvers = rpcerr.re_vers.high; /* highest version supported */
    minvers = rpcerr.re_vers.low;  /* lowest version supported */
    if (RUSERSVERS_ORIG < minvers ||
        RUSERS_ORIG > maxvers) {
        /* doesn't meet minimum standards */
        clnt_perror(clnt, "version mismatch");
        exit(-1);
    }
```

```
    /* This version not supported */
    clnt_destroy(clnt); /* destroy the earlier handle */
    clnt = clnt_create(host, RUSERSPROG,
        RUSERSVERS_ORIG, "udp"); /* try different version */
    if (clnt == NULL) {
        clnt_pcreateerror("clnt");
        exit(-1);
    }
    status = clnt_call(clnt, RUSERSPROCNUM,
        xdr_void, NULL, xdr_u_long, &num_1, to);
    if (status == RPC_SUCCESS) {
        /* We found the latest version number */
        printf("num = %d\n", num_1);
    } else {
        clnt_perror(clnt, "rusers");
        exit(-1);
    }
}
```

TCP

Here is an example that is essentially `rcp`. The initiator of the RPC `snd` call takes its standard input and sends it to the server `rcv`, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
 * The xdr routine:
 *        on decode, read from wire, write onto fp
 *        on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
    XDR *xdrs;
    FILE *fp;
{
    unsigned long size;
    char buf[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE)/* nothing to free */
        return 1;
    while (1) {
        if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread(buf, sizeof(char), BUFSIZ,
                fp)) == 0 && ferror(fp)) {
                fprintf(stderr, "can't fread\n");
                return (1);
            }
        }
        p = buf;
        if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
```

```
            return (0);
        if (size == 0)
            return (1);
        if (xdrs->x_op == XDR_DECODE) {
            if (fwrite(buf, sizeof(char), size,
                fp) != size) {
                fprintf(stderr, "can't fwrite\n");
                return (1);
            }
        }
    }
}
```

```
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include "rcp.h"      /* for prog, vers definitions */

main(argc, argv)
    int argc;
    char **argv;
{
    int xdr_rcp();
    int err;

    if (argc < 2) {
        fprintf(stderr, "usage: %s servername\n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
      RCPVERS, xdr_rcp, stdin, xdr_void, 0) > 0)) {
        clnt_perrno(err);
        fprintf(stderr, "can't make RPC call\n");
        exit(1);
    }
    exit(0);
}

callrpctcp(host, prognum, procnum, versnum,
           inproc, in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
```

```
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "can't get addr for '%s'\n", host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clnttcp_create(&server_addr, prognum,
      versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
        clnt_createerror("rpctcp_create");
        return (-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum,
        inproc, in, outproc, out, total_timeout);
    clnt_destroy(client);
    return ((int)clnt_stat);
}
```

```
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "rcp.h"      /* for prog, vers definitions */

main()
{
    register SVCXPRT *transp;
    int rcp_service(), xdr_rcp();

    if ((transp = svctcp_create(RPC_ANYSOCK,
        BUFSIZ, BUFSIZ)) == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG,
        RCPVERS, rcp_service, IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run();   /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}
```

```
rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (svc_sendreply(transp, xdr_void, 0) == 0)
            fprintf(stderr, "err: rcp_service");
        return;
    case RCPPROC_FP:
        if (!svc_getargs(transp, xdr_rcp, stdout)) {
            svcerr_decode(transp);
            return;
        }
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply\n");
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back to the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an rpc call to the window program, so that it can inform the user that a breakpoint has been reached.

Another case when callback can be useful is when the client cannot block waiting to hear back from the server (possibly because of the huge amount of processing involved in serving the request). In such cases, the server would first acknowledge the request and then use callback to reply.

In order to do an RPC callback, you need a program number to make the RPC call on. Since this will be a dynamically generated program number, it should be in the transient range, `0x40000000 - 0x5fffffff`. The routine `gettransient()` returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the `gettransient()` routine itself. The call to `pmap_set()` is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it.

```
#include <stdio.h>
#include <rpc/rpc.h>
```

```
gettransient(proto, vers, portnum)
    int proto;
    u_long vers;
    u_short portnum;
{
    static u_long prognum = 0x40000000;

    while (!pmap_set(prognum++, vers, proto, portnum))
        continue;
    return (prognum - 1);
}
```

NOTE *The call to* `ntohs()` *for* `portnum` *is not necessary because it was already passed in host byte order (as* `pmap_set()` *expects). See the* `byteorder(3N)` *man page for more details on the conversion of network addresses from network to host byte order.*

The following pair of programs illustrate how to use the `gettransient()` routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program `EXAMPLEPROG`, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an `ALRM` signal in this example), it sends a callback RPC call, using the program number it received earlier.

In this example, both the client and the server are on the same machine. they could very well be on different machines — in that case the handling of the hostname would be different.

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "example.h"

int callback();

main()
{
    int tmp_prog;
    char hostname[256];
    SVCXPRT *xprt;
    int stat;

    gethostname(hostname, sizeof(hostname));
    if ((xprt = svcudp_create(RPC_ANYSOCK)) == NULL) {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }
    if (tmp_prog = gettransient(IPPROTO_UDP, 1,
            xprt->xp_port) == 0) {
```

```
        fprintf(stderr, "failed to get transient number\n");
        exit(1);
    }
    fprintf(stderr, "client gets prognum %d\n", tmp_prog);
    /* protocol is 0 - gettransient does registering */
    (void)svc_register(xprt, tmp_prog, 1, callback, 0);
    stat = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
      EXAMPLEPROC_CALLBACK, xdr_int, &tmp_prog, xdr_void, 0);
    if (stat != RPC_SUCCESS) {
        clnt_perrno(stat);
        exit(1);
    }
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}

callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog\n");
                return (1);
            }
            return (0);
        case 1:
            fprintf(stderr, "client got callback\n");
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog\n");
                return (1);
            }
    }
    return (0);
}
```

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
#include "example.h"

char *getnewprog();
char hostname[256];
int docallback();
int pnum = -1;          /* program number for callback routine */

main()
```

```
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
      EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
    signal(SIGALRM, docallback);
    alarm(10);
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
    int *pnump;
{
    pnum = *(int *)pnump;
    return NULL;
}

docallback()
{
    int ans;

    if (pnum == -1) {
        signal(SIGALRM, docallback);
        return;      /* program number not yet received */
    }
    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
        xdr_void, 0);
    if (ans != RPC_SUCCESS)
        fprintf(stderr, "server: %s\n",clnt_sperrno(ans));
}
```

4.5. Futures

Sun currently supports RPC on top of both UDP (datagram) and TCP (circuit oriented) transports. The RPC library uses sockets API for communicating with the underlying transport layers.

It is likely that in the future releases, the RPC library will use Transport Layer Interface (TLI) API for communicating with the underlying protocol layers. Usage of TLI will help in making RPC transport independent and thus users will be able to use any TLI conforming transport for communication.

Almost all of the current RPC API will be supported. Exceptions would include passing of an open socket to the client and server create routines.

One of the ways to have a very smooth transition to transport independent RPC is to use `rpcgen` to generate the client and the server skeletons, in addition to *not* using any transport specific feature of UDP and TCP. Code written this way will not be bound to run only on UDP and TCP, but will be able to run on all transports of datagram and circuit oriented type. The actual RPC protocol will

however remain the same.

5

External Data Representation: Sun Technical Notes

This chapter contains technical notes on Sun's implementation of the External Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-indcpendent fashion. For a formal specification of the XDR standard, see the *External Data Representation Standard: Protocol Specification*. XDR is the backbone of Sun's Rcmotc Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.[7]

This chapter contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will only need the information in the *Number Filters*, *Floating Point Filters*, and *Enumeration Filters* sections. Programmers wishing to implement RPC and XDR on new machines will be interested in the rest of the chapter, as well as the *External Data Representation Standard: Protocol Specification*, which will be their primary reference.

NOTE `rpcgen` *can be used to write XDR routines even in cases where no RPC calls are being made.*

On Sun systems, C programs that want to use XDR routines must include the filc `<rpc/rpc.h>`, which contains all the necessary interfaces to the XDR system. Since the C library `libc.a` contains all the XDR routines, compile as normal.

```
example% cc program.c
```

[7] For a complete specification of the system External Data Representation routines, see the `xdr(3N)` manual page.

Justification

Consider the following two programs, `writer`:

```
#include <stdio.h>

main()          /* writer.c */
{
    long i;

    for (i = 0; i < 8; i++) {
        if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) (
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

and `reader`:

```
#include <stdio.h>

main()          /* reader.c */
{
    long i, j;

    for (j = 0; j < 8; j++) (
        if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        )
        printf("%ld ", i);
    )
    printf("\n");
    exit(0);
)
```

The two programs appear to be portable, because (a) they pass `lint` checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%


vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and 4.2BSD came the concept of "network pipes" — a process produces data on one machine, and a second process

consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()`, a filter that knows the standard representation of a long integer in its external form. Here are the revised versions of `writer`:

```
#include <stdio.h>
#include <rpc/rpc.h>      /* xdr is a sub-library of rpc */

main()        /* writer.c */
{
    XDR xdrs;
    long i;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

and `reader`:

```
#include <stdio.h>
#include <rpc/rpc.h>     /* xdr is a sub-library of rpc */

main()       /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%

sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

NOTE *Integers are just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.*

A Canonical Standard

XDR's approach to standardizing data representations is *canonical*. That is, XDR defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representaions to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable data

creators and users. A new machine joins this community by being "taught" how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

The XDR Library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use `xdrstdio_create()`. The parameters to XDR stream creation routines vary

according to their function. In our example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a `FILE` that the input or output is performed on, and the operation. The operation may be `XDR_ENCODE` for serializing in the `writer` program, or `XDR_DECODE` for deserializing in the `reader` program.

Note: RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns `FALSE` (0) if it fails, and `TRUE` (1) if it succeeds. Second, for each data type, `xxx`, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, xp)
    XDR *xdrs;
    xxx *xp;
{
}
```

In our case, `xxx` is long, and the corresponding XDR routine is a primitive, `xdr_long()`. The client could also define an arbitrary structure `xxx` in which case the client would also supply the routine `xdr_xxx()`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, `xdrs` can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. See the *XDR Operation Directions* section of this chapter for details.

Let's look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t          /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities))
        return(TRUE);
    return(FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return `FALSE` if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are `TRUE` (1) and `FALSE` (0). This document uses the following definitions:

```
#define bool_t  int
#define TRUE    1
#define FALSE   0
```

Keeping these conventions in mind, `xdr_gnumbers()` can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

5.1. XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, automatically included by `<rpc/rpc.h>`.

Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, cp)
    XDR *xdrs;
    char *cp;

bool_t xdr_u_char(xdrs, ucp)
    XDR *xdrs;
    unsigned char *ucp;

bool_t xdr_int(xdrs, ip)
    XDR *xdrs;
    int *ip;

bool_t xdr_u_int(xdrs, up)
    XDR *xdrs;
    unsigned *up;

bool_t xdr_long(xdrs, lip)
    XDR *xdrs;
    long *lip;

bool_t xdr_u_long(xdrs, lup)
    XDR *xdrs;
    u_long *lup;

bool_t xdr_short(xdrs, sip)
    XDR *xdrs;
    short *sip;

bool_t xdr_u_short(xdrs, sup)
    XDR *xdrs;
    u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return `TRUE` if they complete successfully, and `FALSE` otherwise.

Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
    XDR *xdrs;
    float *fp;

bool_t xdr_double(xdrs, dp)
    XDR *xdrs;
    double *dp;
```

The first parameter, `xdrs` is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return `TRUE` if they complete successfully, and `FALSE` otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C `enum` has the same representation inside the machine as a C integer. The boolean type is an important instance of the `enum`. The external representation of a boolean is always `TRUE` (1) or `FALSE` (0).

```
#define bool_t  int
#define FALSE   0
#define TRUE    1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
    XDR *xdrs;
    enum_t *ep;

bool_t xdr_bool(xdrs, bp)
    XDR *xdrs;
    bool_t *bp;
```

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`.

No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();   /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions then the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, `XDR_FREE`. To review, the three XDR directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
    XDR *xdrs;
    char **sp;
    u_int maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding. its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters.

Keep `maxlength` small. If it is too big you can blow the heap, since `xdr_string()` will call `malloc()` for space.

The routine returns `FALSE` if the number of characters exceeds `maxlength`, and `TRUE` if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction `XDR_ENCODE` is easiest to understand. The parameter `sp` points to a string of a certain length; if the string does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the value is `NULL`, then a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-null, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than `maxlength`. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the `XDR_FREE` operation, the string is obtained by dereferencing `sp`. If the string is not `NULL`, it is freed and `*sp` is set to `NULL`. In this operation, `xdr_string()` ignores the `maxlength` parameter.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
    XDR *xdrs;
    char **bpp;
    u_int *lp;
    u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of `xdr_string()`, respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsiz;
    bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is `NULL` when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsiz` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The `xdr_element()` routine is called to serialize, deserialize, or free each element of the array.

Before defining more constructed data types, it is appropriate to present three examples.

Example A:
A user on a networked machine can be identified by (a) the machine name, such as `krypton`: see the `gethostname` man page; (b) the user's UID: see the `geteuid` man page; and (c) the group numbers to which the user belongs: see the `getgroups` man page. A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
    char    *nu_machinename;
    int     nu_uid;
    u_int   nu_glen;
    int     *nu_gids;
};
#define NLEN 255     /* machine names < 256 chars */
#define NGRPS 20     /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
        xdr_int(xdrs, &nup->nu_uid) &&
        xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
        NGRPS, sizeof (int), xdr_int));
}
```

Example B:
A party of network users could be implemented as an array of `netuser` structure. The declaration and its associated XDR routines are as follows:

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
        sizeof (struct netuser), xdr_netuser));
}
```

Example C:
The well-known parameters to `main`, `argc` and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};
#define ALEN 1000    /* args cannot be > 1000 chars */
#define NARGC 100    /* commands cannot have > 100 args */

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_wrapstring(xdrs, sp)
    XDR *xdrs;
    char **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
    XDR *xdrs;
    struct cmd *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrapstring));
}

bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
    struct history *hp;
{
    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
        sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrapstring()` is needed to package the `xdr_string()` routine, because the implementation of `xdr_array()` only passes two parameters to the array element description routine; `xdr_wrapstring()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The `xdr_opaque()` primitive is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
    XDR *xdrs;
    char *p;
    u_int len;
```

The parameter p is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

Fixed Sized Arrays

The XDR library provides a primitive, `xdr_vector()`, for fixed-length arrays.

```
#define NLEN 255     /* machine names must be < 256 chars */
#define NGRPS 20     /* user belongs to exactly 20 groups */

struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    int i;

    if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
        return(FALSE);
    if (!xdr_int(xdrs, &nup->nu_uid))
        return(FALSE);
    if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
        xdr_int)) {
            return(FALSE);
    }
    return(TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an "arm" of the union.

```
struct xdr_discrim {
    enum_t value;
    bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
    XDR *xdrs;
    enum_t *dscmp;
    char *unp;
    struct xdr_discrim *arms;
    bool_t (*defaultarm)();   /* may equal NULL */
```

First the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always an `enum_t`. Next the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of `[value,proc]`. If the union's discriminant is equal to the associated `value`, then the `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL` (0). If the discriminant is not found in the `arms` array, then the `defaultarm` procedure is called if it is non-null; otherwise the routine returns `FALSE`.

Example D: Suppose the type of a union may be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
    enum utype utype;    /* the union's discriminant */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrapstring },
    { __dontcare__, NULL }
    /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return(xdr_union(xdrs, &utp->utype, &utp->uval,
        u_tag_arms, NULL));
}
```

The routine `xdr_gnumbers()` was presented above in *The XDR Library* section. `xdr_wrapstring()` was presented in example C. The default `arm` parameter to `xdr_union()` (the last parameter) is `NULL` in this example. Therefore the value of the union's discriminant may legally take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise: Implement `xdr_union()` using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The `xdr_reference()` primitive makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
    XDR *xdrs;
    char **pp;
    u_int ssize;
    bool_t (*proc)();
```

Parameter `pp` is the address of the pointer to the structure; parameter `ssize` is the size in bytes of the structure (use the C function `sizeof()` to obtain this value); and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is `NULL`.

There is no need for a primitive `xdr_struct()` to describe structures within structures, because pointers are always sufficient.

Exercise: Implement `xdr_reference()` using `xdr_array()`. Warning: `xdr_reference()` and `xdr_array()` are NOT interchangeable external representations of data.

Example E: Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
      xdr_reference(xdrs, &pp->gnp,
      sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

Pointer Semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value `NULL` (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E a `NULL` pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a null-value pointer during serialization. That is, passing an address of a pointer whose value is `NULL` to `xdr_reference()` when serialing data will most likely cause a memory fault and, on the UNIX system, a core dump.

`xdr_pointer()` correctly handles `NULL` pointers. For more information about its use, see *Linked Lists*.

Exercise: After reading the section on *Linked Lists*, return here and extend example E so that it can correctly deal with `NULL` pointer values.

Exercise: Using the `xdr_union()`, `xdr_reference()` and `xdr_void()` primitives, implement a generic pointer handling primitive that implicitly deals

with NULL pointers. That is, implement `xdr_pointer()`.

Non-filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
    XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
    XDR *xdrs;
    u_int pos;

xdr_destroy(xdrs)
    XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream. Warning: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`. Warning: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return `FALSE`. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

XDR Operation Directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation — `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`. The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in the *Linked Lists* section, below, demonstrates the usefulness of the `xdrs->x_op` field.

XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O `FILE` streams, TCP/IP connections and UNIX files, and memory.

Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>     /* xdr streams part of rpc */

void
xdrstdio_create(xdrs, fp, x_op)
    XDR *xdrs;
    FILE *fp;
    enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
    XDR *xdrs;
    char *addr;
    u_int len;
    enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `sendto()` system routine.

Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>     /* xdr streams part of rpc */

xdrrec_create(xdrs,
  sendsize, recvsize, iohandle, readproc, writeproc)
    XDR *xdrs;
    u_int sendsize, recvsize;
    char *iohandle;
    int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine `readproc()` or `writeproc()` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters (`buf` and `nbytes`) and the results (byte count) are identical to the system routines. If `xxx` is `readproc()` or `writeproc()`, then it has the following form:

```
/*
 * returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
    char *iohandle;
    char *buf;
    int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in the *Advanced Topics* section, below. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
    XDR *xdrs;
    bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
    XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
    XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is `TRUE`, then the stream's `writeproc` will be called; otherwise, `writeproc` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns `TRUE`. That is not to say that there is no more data in the underlying file descriptor.

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };

typedef struct {
    enum xdr_op x_op;     /* operation; fast added param */
    struct xdr_ops {
        bool_t   (*x_getlong)();   /* get long from stream */
        bool_t   (*x_putlong)();   /* put long to stream */
        bool_t   (*x_getbytes)();  /* get bytes from stream */
        bool_t   (*x_putbytes)();  /* put bytes to stream */
        u_int    (*x_getpostn)();  /* return stream offset */
        bool_t   (*x_setpostn)();  /* reposition offset */
        caddr_t  (*x_inline)();    /* ptr to buffered data */
        VOID     (*x_destroy)();   /* free private area */
    } *x_ops;
    caddr_t x_public;     /* users' data */
    caddr_t x_private;    /* pointer to private data */
    caddr_t x_base;       /* private for position info */
    int     x_handy;      /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. `x_getpostn()`, `x_setpostn()`, and `x_destroy()`, are macros for accessing operations. The operation `x_inline()` takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return `NULL` if it cannot return a buffer segment of the requested size. (The `x_inline()` routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream; they return `TRUE` if they are successful, and `FALSE` otherwise. The routines have identical parameters (replace `xxx`):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
    XDR *xdrs;
    char *buf;
    u_int bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return *TRUE* if they succeed, and `FALSE` otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
    XDR *xdrs;
    long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

5.2. Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. The *External Data Representation Standard: Protocol Specification* chapter of this *Network Programming* manual describes this language in complete detail.

Linked Lists

The last example in the *Pointers* section presented a C data structure and its associated XDR routines for a individual's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &(gp->g_assets)))
        return(xdr_long(xdrs, &(gp->g_liabilities)));
    return(FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `gn_next` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`:

```
struct gnumbers {
    int g_assets;
    int g_liabilities;
};

struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In this description, the boolean indicates whether there is more data following it. If the boolean is `FALSE`, then it is the last data field of the structure. If it is `TRUE`, then it is followed by a gnumbers structure and (recursively) by a `gnumbers_list`. Note that the C declaration has no boolean explicitly declared in it (though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow easily from the XDR description above. Note how the primitive `xdr_pointer()` is used to implement the XDR union above.

```
bool_t
xdr_gnumbers_node(xdrs, gn)
    XDR *xdrs;
    gnumbers_node *gn;
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp,
        sizeof(struct gnumbers_node),
        xdr_gnumbers_node));
}
```

The unfortunate side effect of XDR'ing a list with these routines is that the C stack grows linearly with respect to the number of node in the list. This is due to the recursion. The following routine collapses the above two mutually recursive into a single, non-recursive one.

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    bool_t more_data;
    gnumbers_list *nextp;

    for (;;) {
        more_data = (*gnp != NULL);
        if (!xdr_bool(xdrs, &more_data)) {
            return(FALSE);
        }
        if (! more_data) {
            break;
        }
        if (xdrs->x_op == XDR_FREE) {
            nextp = &(*gnp)->gn_next;
        }
        if (!xdr_reference(xdrs, gnp,
            sizeof(struct gnumbers_node), xdr_gnumbers)) {

        return(FALSE);
        }
        gnp = (xdrs->x_op == XDR_FREE) ?
            nextp : &(*gnp)->gn_next;
    }
    *gnp = NULL;
    return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that this boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until we deserialize it in the next statement.

The next statement XDR's the more_data field of the XDR union. Then if there is truly no more data, we set this last pointer to NULL to indicate the end of the list, and return TRUE because we are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

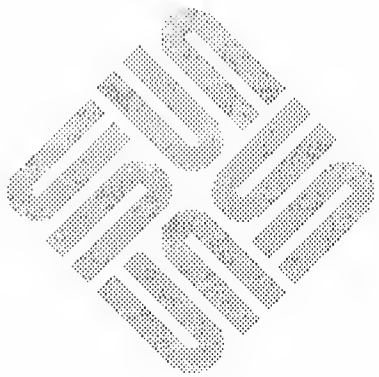
Next, if the direction is XDR_FREE, the value of nextp is set to indicate the location of the next pointer in the list. We do this now because we need to dereference gnp to find the location of the next item in the list, and after the next statement the storage pointed to by gnp will be freed up and no be longer valid. We can't do this for all directions though, because in the XDR_DECODE direction the value of gnp won't be set until the next statement.

Next, we XDR the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer() which we used before, but it does not send over the boolean indicating whether there is more data. We use it

instead of `xdr_pointer()` because we have already XDR'd this information ourselves. Notice that the xdr routine passed is not the same type as an element in the list. The routine passed is `xdr_gnumbers()`, for XDR'ing gnumbers, but each element in the list is actually of type `gnumbers_node`. We don't pass `xdr_gnumbers_node()` because it is recursive, and instead use `xdr_gnumbers()` which XDR's all of the non-recursive part. Note that this trick will work only if the `gn_numbers` field is the first item in each element, so that their addresses are identical when passed to `xdr_reference()`.

Finally, we update *gnp* to point to the next item in the list. If the direction is `XDR_FREE`, we set it to the previously saved value, otherwise we can dereference *gnp* to get the proper value. Though harder to understand than the recursive version, this non-recursive routine is far less likely to blow the C stack. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.

PART TWO: Protocol Specifications



6

External Data Representation Standard: Protocol Specification

6.1. Status of this Standard

Note: This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It has been designated RFC1014 by the ARPA Network Information Center.

6.2. Introduction

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the Sun Workstation, VAX, IBM-PC, and Cray. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language [1], just as Courier [4] is similar to Mesa. Protocols such as Sun RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

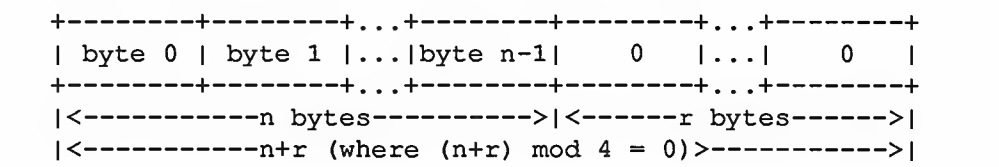
Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more

additional bytes where required.

A Block



6.3. XDR Data Types

Each of the sections that follow describes a data type defined in the XDR standard, shows how it is declared in the language, and includes a graphic illustration of its encoding.

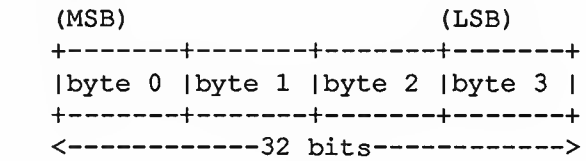
For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. "n", "m" and "r" denote integers. For the full language specification and more formal definitions of terms such as "identifier" and "declaration", refer to *The XDR Language Specification*, below.

For some data types, more specific examples are included. A more extensive example of a data description is in *An Example of an XDR Data Description*, below.

Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

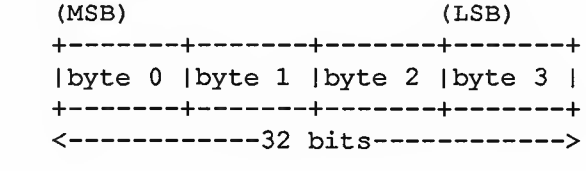
Integer



Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

Unsigned Integer



Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... } identifier;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

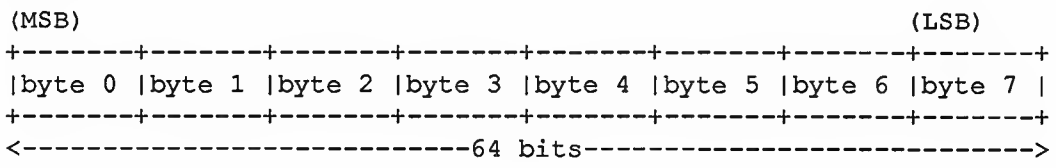
This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Hyper Integer and Unsigned Hyper Integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:

Hyper Integer
Unsigned Hyper Integer



Floating-point

The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [3]. The following three fields describe the single-precision floating-point number:

- **S**: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.
- **E**: The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.
- **F**: The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

Single-Precision Floating-Point

```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
S|   E   |           F           |
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating- point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

Double-precision Floating-point

The standard defines the encoding for the double-precision floating- point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers [3]. The standard encodes the following three fields, which describe the double-precision floating-point number:

- **S**: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.
- **E**: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.
- **F**: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

Double-Precision Floating-Point

```
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
S|    E   |                      F                       |
+------+------+------+------+------+------+------+------+
1|<--11-->|<-----------------52 bits------------------->|
<-----------------------64 bits------------------------->
```

Just as the most and least significant bytes of a number are 0 and 3, the most and

least significant bits of a double-precision floating- point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

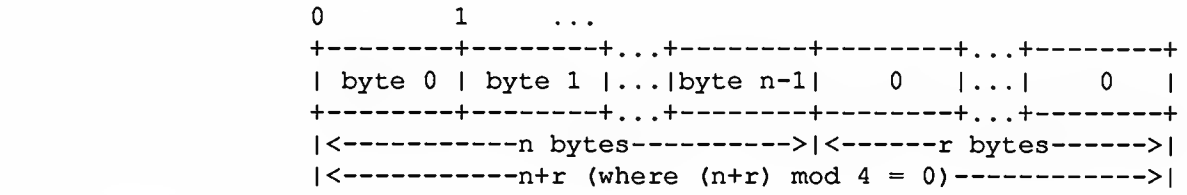
Fixed-length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called "opaque" and is declared as follows:

```
opaque identifier[n];
```

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

Fixed-Length Opaque



Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

or

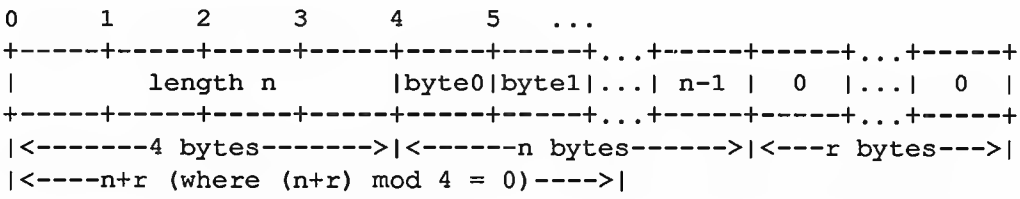
```
opaque identifier<>;
```

The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

This can be illustrated as follows:

Variable-Length Opaque



It is an error to encode a length greater than the maximum described in the specification.

String

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
string object<m>;
```

or

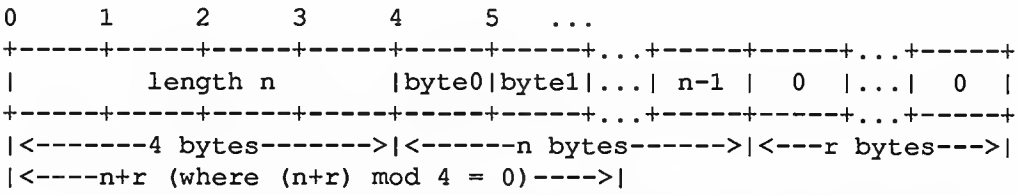
```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
string filename<255>;
```

Which can be illustrated as:

A String



It is an error to encode a length greater than the maximum described in the specification.

Fixed-length Array

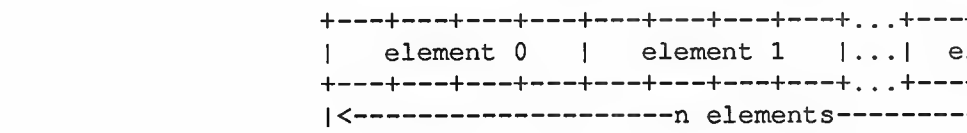
Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n-1 are encoded by

individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type "string", yet each element will vary in its length.

Fixed-Length Array



Variable-length Array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1. The declaration for variable-length arrays follows this form:

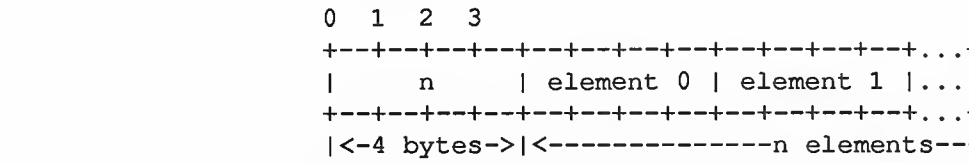
```
type-name identifier<m>;
```

or

```
type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be (2**32) - 1.

Counted Array



It is an error to encode a value of n that is greater than the maximum described in the specification.

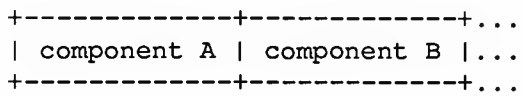
Structure

Structures are declared as follows:

```
struct {
    component-declaration-A;
    component-declaration-B;
    ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

Structure



Discriminated Union

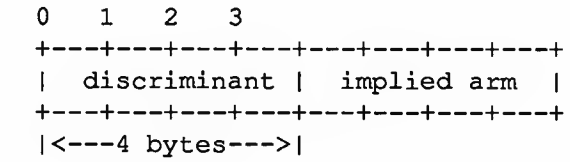
A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either "int", "unsigned int", or an enumerated type, such as "bool". The component types are called "arms" of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
    case discriminant-value-A:
    arm-declaration-A;
    case discriminant-value-B:
    arm-declaration-B;
    ...
    default: default-declaration;
} identifier;
```

Each "case" keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated Union

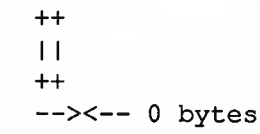


Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:



Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

"const" is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

Typedef

"typedef" does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called "eggbox" using an existing type called "egg":

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the typedef, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable "fresheggs":

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the "typedef" part and placing the identifier after the "struct", "union", or "enum" keyword, instead of at the end. For example, here are the two ways to define the type "bool":

```
typedef enum {      /* using typedef */
    FALSE = 0,
    TRUE = 1
    } bool;

enum bool {         /* preferred alternative */
    FALSE = 0,
    TRUE = 1
    };
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
    case TRUE:
    type-name element;
    case FALSE:
    void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean "opted" can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type "stringlist" that encodes lists of arbitrary length strings:

```
struct *stringlist {
    string item<>;
    stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
    case TRUE:
        struct {
            string item<>;
            stringlist next;
        } element;
    case FALSE:
        void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
    string item<>;
    stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

Areas for Future Enhancement

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C so that applications written in these languages will be able to communicate easily over some medium.

One could imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this are support for different block sizes and byte-orders. The XDR discussed here could then be considered the 4-byte big-endian member of a larger XDR family.

6.4. Discussion

Why a Language for Describing Data?

There are many advantages in using a data-description language such as XDR versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand and allow one to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

Why Only one Byte-Order for an XDR Unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, though, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it, since no higher level protocol is necessary for determining the byte-order.

Why does XDR use Big-Endian Byte-Order?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the Motorola 68000 and IBM 370, support the big-endian byte-order.

Why is the XDR Unit Four Bytes Wide?

There is a tradeoff in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. We chose four as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

Why must Variable-Length Data be Padded with Zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

Why is there No Explicit Data-Typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

6.5. The XDR Language Specification

Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

1. The characters `|`, `(`, `)`, `[`, `]`, `,` and * are special.
2. Terminal symbols are strings of any characters surrounded by double quotes.
3. Non-terminal symbols are strings of non-special characters.
4. Alternative items are separated by a vertical bar (`|`).
5. Optional items are enclosed in brackets.
6. Items are grouped together by enclosing them in parentheses.
7. A * following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

```
"a " "very" (", " " very")* [" cold " "and"] " rainy " ("day" | "night")
```

An infinite number of strings match this pattern. A few of them are:

```
"a very rainy day"
"a very, very rainy day"
"a very cold and rainy day"
"a very, very, very cold and rainy night"
```

Lexical Notes

1. Comments begin with '/*' and terminate with '*/'.
2. White space serves to separate items and is otherwise ignored.
3. An identifier is a letter followed by an optional sequence of letters, digits or underbar ('_'). The case of identifiers is not ignored.

4. A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ('-').

Syntax Information

```
declaration:
    type-specifier identifier
    | type-specifier identifier "[" value "]"
    | type-specifier identifier "<" [ value ] ">"
    | "opaque" identifier "[" value "]"
    | "opaque" identifier "<" [ value ] ">"
    | "string" identifier "<" [ value ] ">"
    | type-specifier "*" identifier
    | "void"

value:
    constant
    | identifier

type-specifier:
      [ "unsigned" ] "int"
    | [ "unsigned" ] "hyper"
    | "float"
    | "double"
    | "bool"
    | enum-type-spec
    | struct-type-spec
    | union-type-spec
    | identifier

enum-type-spec:
    "enum" enum-body

enum-body:
    "{"
    ( identifier "=" value )
    ( "," identifier "=" value )*
    "}"

struct-type-spec:
    "struct" struct-body

struct-body:
    "{"
    ( declaration ";" )
    ( declaration ";" )*
    "}"

union-type-spec:
    "union" union-body

union-body:
    "switch" "(" declaration ")" "{"
    ( "case" value ":" declaration ";" )
    ( "case" value ":" declaration ";" )*
```

```
    [ "default" ":" declaration ";" ]
    "}"

constant-def:
    "const" identifier "=" constant ";"

type-def:
    "typedef" declaration ";"
    | "enum" identifier enum-body ";"
    | "struct" identifier struct-body ";"
    | "union" identifier union-body ";"

definition:
    type-def
    | constant-def

specification:
    definition *
```

Syntax Notes

1. The following are keywords and cannot be used as identifiers: "bool", "case", "const", "default", "double", "enum", "float", "hyper", "opaque", "string", "struct", "switch", "typedef", "union", "unsigned" and "void".
2. Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a "const" definition.
3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
4. Similarly, variable names must be unique within the scope of struct and union declarations. Nested struct and union declarations create new scopes.
5. The discriminant of a union must be of a type that evaluates to an integer. That is, "int", "unsigned int", "bool", an enumerated type or any typedefed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

6.6. An Example of an XDR Data Description

Here is a short XDR data description of a thing called a "file", which might be used to transfer files from one machine to another.

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file     */
const MAXNAMELEN = 255;     /* max length of a file name */

/*
 * Types of files:
 */

enum filekind {
    TEXT = 0,         /* ascii data */
    DATA = 1,         /* raw data   */
    EXEC = 2          /* executable */
};

/*
 * File information, per kind of file:
 */

union filetype switch (filekind kind) {
    case TEXT:
        void;                                /* no extra information */
    case DATA:
        string creator<MAXNAMELEN>;          /* data creator      */
    case EXEC:
        string interpretor<MAXNAMELEN>;      /* program interpretor */
};

/*
 * A complete file:
 */

struct file {
    string filename<MAXNAMELEN>;  /* name of file */
    filetype type;                /* info about file */
    string owner<MAXUSERNAME>;    /* owner of file  */
    opaque data<MAXFILELEN>;      /* file data      */
};
```

Suppose now that there is a user named "john" who wants to store his lisp program "sillyprog" that contains just the data "(quit)". His file would be encoded as follows:

Offset	*Hex Bytes*	*ASCII*	*Description*
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpretor = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

6.7. References

[1] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language", Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE Computer, October 1981.

[3] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[4] "Courier: The Remote Procedure Call Protocol", XEROX Corporation, XSIS 038112, December 1981.

7

Remote Procedure Calls: Protocol Specification

7.1. Status of this Memo

Note: This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It has been designated RFC 1050 by the ARPA-Internet Network Information Center.

7.2. Introduction

This chapter specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package. (The message protocol is specified with the External Data Representation (XDR) language. See the *External Data Representation Standard: Protocol Specification* for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify it or its uses). The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.

Terminology

This chapter discusses servers, services, programs, procedures, clients, and versions. A server is a piece of software where network services are implemented. A network service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification (see the *Port Mapper Program Protocol*, below, for an example). Network clients are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file IO and have procedures like "read" and "write". A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes—one is the caller's process, the other is a server's

process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this model, only one of the two processes is active at any given time. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but

if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP[2] is perhaps the most natural transport for RPC.

NOTE *At Sun, RPC is currently implemented on top of both TCP/IP and UDP/IP transports.*

Binding and Rendezvous Independence

The act of binding a client to a service is NOT part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself—see the *Port Mapper Program Protocol*, below).

Implementors should think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in the *Authentication Protocols*, below.

7.3. RPC Protocol Requirements

The RPC protocol must provide for the following:

1. Unique specification of a procedure to be called.
2. Provisions for matching response messages to request messages.
3. Provisions for authenticating the caller to service and vice-versa.

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

1. RPC protocol mismatches.
2. Remote program protocol version mismatches.
3. Protocol errors (such as misspecification of a procedure's parameters).
4. Reasons why remote authentication failed.
5. Any other reasons why the desired procedure was not called.

Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (like Sun). Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is "read" and procedure number 12 is "write".

Just as remote program protocols may change over several versions, the actual RPC message protocol could also change. Therefore, the call message also has in it the RPC version number, which is always equal to two for the version of RPC described here.

The reply message to a request message has enough information to distinguish the following error conditions:

1. The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
2. The remote program is not available on the remote system.
3. The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
4. The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
5. The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

Authentication

Provisions for authentication of caller to service and vice-versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
    AUTH_NULL   = 0,
    AUTH_UNIX   = 1,
    AUTH_SHORT  = 2,
    AUTH_DES    = 3
    /* and more to be defined */
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

In simple English, any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See *Authentication Protocols*, below, for definitions of the various authentication protocols.)

If authentication parameters were rejected, the response message contains information stating why they were rejected.

Program Number Assignment

Program numbers are given out in groups of `0x20000000` (decimal 536870912) according to the following chart:

Program Numbers	*Description*
0 - 1fffffff	*Defined by Sun*
20000000 - 3fffffff	*Defined by user*
40000000 - 5fffffff	*Transient*
60000000 - 7fffffff	*Reserved*
80000000 - 9fffffff	*Reserved*
a0000000 - bfffffff	*Reserved*
c0000000 - dfffffff	*Reserved*
e0000000 - ffffffff	*Reserved*

The first group is a range of numbers administered by Sun Microsystems and should be identical for all sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Sun currently uses, or perhaps abuses, the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC.

These two protocols are discussed but not defined below.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP/IP) for its transport. In the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgement).

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See the *Port Mapper Program Protocol*, below, for more information.

7.4. The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style.

```
enum msg_type {
    CALL  = 0,
    REPLY = 1
};

/*
* A reply to a call message can take on two forms:
* The message was either accepted or rejected.
*/
enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED   = 1
};

/*
* Given that a call message was accepted, the following is the
* status of an attempt to call a remote procedure.
*/
enum accept_stat {
    SUCCESS       = 0, /* RPC executed successfully     */
    PROG_UNAVAIL  = 1, /* remote hasn't exported program */
    PROG_MISMATCH = 2, /* remote can't support version # */
    PROC_UNAVAIL  = 3, /* program can't support procedure */
    GARBAGE_ARGS  = 4  /* procedure can't decode params  */
};

/*
* Reasons why a call message was rejected:
*/
enum reject_stat {
    RPC_MISMATCH = 0, /* RPC version number != 2       */
    AUTH_ERROR = 1    /* remote can't authenticate caller */
```

```
};

/*
* Why authentication failed:
*/
enum auth_stat {
    AUTH_BADCRED       = 1,   /* bad credentials */
    AUTH_REJECTEDCRED  = 2,   /* client must begin new session */
    AUTH_BADVERF       = 3,   /* bad verifier */
    AUTH_REJECTEDVERF  = 4,   /* verifier expired or replayed */
    AUTH_TOOWEAK       = 5    /* rejected for security reasons */
};

/*
* The RPC message:
* All messages start with a transaction identifier, xid,
* followed by a two-armed discriminated union. The union's
* discriminant is a msg_type which switches to one of the two
* types of the message. The xid of a REPLY message always
* matches that of the initiating CALL message. NB: The xid
* field is only used for clients matching reply messages with
* call messages or for servers detecting retransmissions; the
* service side cannot treat this id as any type of sequence
* number.
*/
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

/*
* Body of an RPC request call:
* In version 2 of the RPC protocol specification, rpcvers must
* be equal to 2. The fields prog, vers, and proc specify the
* remote program, its version number, and the procedure within
* the remote program to be called. After these fields are two
* authentication parameters: cred (authentication credentials)
* and verf (authentication verifier). The two authentication
* parameters are followed by the parameters to the remote
* procedure, which are specified by the specific program
* protocol.
*/
struct call_body {
    unsigned int rpcvers;   /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
```

```
    opaque_auth verf;
    /* procedure specific parameters start here */
};

/*
* Body of a reply to an RPC request:
* The call message was either accepted or rejected.
*/
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;

/*
* Reply to an RPC request that was accepted by the server:
* there could be an error even though the request was accepted.
* The first field is an authentication verifier that the server
* generates in order to validate itself to the caller. It is
* followed by a union whose discriminant is an enum
* accept_stat. The SUCCESS arm of the union is protocol
* specific. The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP
* arms of the union are void. The PROG_MISMATCH arm specifies
* the lowest and highest version numbers of the remote program
* supported by the server.
*/
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            opaque results[0];
            /* procedure-specific results start here */
        case PROG_MISMATCH:
            struct {
                unsigned int low;
                unsigned int high;
            } mismatch_info;
        default:
            /*
            * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL,
            * and GARBAGE_ARGS.
            */
            void;
    } reply_data;
};

/*
* Reply to an RPC request that was rejected by the server:
* The request can be rejected for two reasons: either the
* server is not running a compatible version of the RPC
* protocol (RPC_MISMATCH), or the server refuses to
* authenticate the caller (AUTH_ERROR). In case of an RPC
```

```
* version mismatch, the server returns the lowest and highest
* supported RPC version numbers. In case of refused
* authentication, failure status is returned.
*/
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

7.5. Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication implemented at (and supported by) Sun. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

Null Authentication

Often calls must be made where the caller does not know who he is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the `opaque_auth`'s union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL`. The bytes of the opaque_auth's body are undefined. It is recommended that the opaque length be zero.

UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the credential's discriminant of an RPC call message is `AUTH_UNIX`. The bytes of the credential's opaque body encode the following structure:

```
struct auth_unix {
    unsigned int stamp;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<10>;
};
```

The `stamp` is an arbitrary ID which the caller machine may generate. The `machinename` is the name of the caller's machine (like "krypton"). The `uid` is the caller's effective user ID. The `gid` is the caller's effective group ID. The `gids` is a counted array of groups which contain the caller as a member. The verifier accompanying the credentials should be of `AUTH_NULL` (defined above).

The value of the discriminant of the response verifier received in the reply message from the server may be `AUTH_NULL` or `AUTH_SHORT`. In the case of `AUTH_SHORT`, the bytes of the response verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original `AUTH_UNIX` flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back by way of an `AUTH_SHORT` style

response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be `AUTH_REJECTEDCRED`. At this point, the caller may wish to try the original `AUTH_UNIX` style of credentials.

DES Authentication

UNIX authentication suffers from two major problems:

1. The naming is too UNIX-system oriented.
2. There is no verifier, so credentials can easily be faked.

DES authentication attempts to fix these two problems.

Naming

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the "netname" or network name of the caller. The server is not allowed to interpret the contents of the caller's name in any other way except to identify the caller. Thus, netnames should be unique for every caller in the internet.

It is up to each operating system's implementation of DES authentication to generate netnames for its users that insure this uniqueness when they call upon remote servers. Operating systems already know how to distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a UNIX user at Sun with a user ID of 515 might be assigned the following netname: "unix.515@sun.com". This netname contains three items that serve to insure it is unique. Going backwards, there is only one naming domain called "sun.com" in the internet. Within this domain, there is only one UNIX user with user ID 515. However, there may be another user on another operating system, for example VMS, within the same naming domain that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we add the operating system name. So one user is "unix.515@sun.com" and the other is "vms.515@sun.com".

The first field is actually a naming method rather than an operating system name. It just happens that today there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be the name of that standard, instead of an operating system name.

DES Authentication Verifiers

Unlike UNIX authentication, DES authentication does have a verifier so the server can validate the client's credential (and vice-versa). The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to what the real time is, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the "conversation key" of the RPC session. And if the client knows the conversation key, then it must be the real client.

The conversation key is a DES [5] key which the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in DES authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time in order for all of this to work. If network time synchronization cannot be guaranteed, then the client can synchronize with the server before beginning the conversation, perhaps by consulting the Internet Time Server (TIME[4]).

The way a server determines if a client timestamp is valid is somewhat complicated. For any other transaction but the first, the server just checks for two things:

1. the timestamp is greater than the one previously seen from the same client.
2. the timestamp has not expired.

A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's "window". The "window" is a number the client passes (encrypted) to the server in its first transaction. You can think of it as a lifetime for the credential.

This explains everything but the first transaction. In the first transaction, the server checks only that the timestamp has not expired. If this was all that was done though, then it would be quite easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. As an added check, the client sends an encrypted item in the first transaction known as the "window verifier" which must be equal to the window minus 1, or the server will reject the credential.

The client too must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything different than this, it will reject it.

Nicknames and Clock Synchronization

After the first transaction, the server's DES authentication subsystem returns in its verifier to the client an integer "nickname" which the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server which stores for each client its netname, decrypted DES key and window.

Though they originally were synchronized, the client's and server's clocks can get out of sync again. When this happens the client RPC subsystem most likely will get back `RPC_AUTHERROR` at which point it should resynchronize.

A client may still get the `RPC_AUTHERROR` error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. A client should resend its original credential in this case and the server will give it a new nickname. If a server crashes, the entire nickname table gets flushed, and all clients will have to resend their original credentials.

DES Authentication Protocol (in XDR language)

```
/*
* There are two kinds of credentials: one in which the client uses
* its full network name, and one in which it uses its "nickname"
* (just an unsigned integer) given to it by the server. The
* client must use its fullname in its first transaction with the
* server, in which the server will return to the client its
* nickname. The client may use its nickname in all further
* transactions with the server. There is no requirement to use the
* nickname, but it is wise to use it for performance reasons.
*/
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};

/*
* A 64-bit block of encrypted DES data
*/
typedef opaque des_block[8];

/*
* Maximum length of a network user's name
*/
const MAXNETNAMELEN = 255;

/*
* A fullname contains the network name of the client, an encrypted
* conversation key and the window. The window is actually a
* lifetime for the credential. If the time indicated in the
* verifier timestamp plus the window has past, then the server
* should expire the request and not grant it. To insure that
* requests are not replayed, the server should insist that
* timestamps are greater than the previous one seen, unless it is
* the first transaction. In the first transaction, the server
* checks instead that the window verifier is one less than the
* window.
*/
struct authdes_fullname {
string name<MAXNETNAMELEN>;   /* name of client */
des_block key;                /* PK encrypted conversation key */
unsigned int window;          /* encrypted window */
};

/*
* A credential is either a fullname or a nickname
*/
union authdes_cred switch (authdes_namekind adc_namekind) {
    case ADN_FULLNAME:
        authdes_fullname adc_fullname;
    case ADN_NICKNAME:
        unsigned int adc_nickname;
};
```

```
/*
* A timestamp encodes the time since midnight, January 1, 1970.
*/
struct timestamp {
    unsigned int seconds;      /* seconds */
    unsigned int useconds;     /* and microseconds */
};

/*
* Verifier: client variety
* The window verifier is only used in the first transaction. In
* conjunction with a fullname credential, these items are packed
* into the following structure before being encrypted:
*
* struct {
*   adv_timestamp;          -- one DES block
*   adc_fullname.window;  -- one half DES block
*   adv_winverf;        -- one half DES block
* }
* This structure is encrypted using CBC mode encryption with an
* input vector of zero. All other encryptions of timestamps use
* ECB mode encryption.
*/
struct authdes_verf_clnt {
    timestamp adv_timestamp;      /* encrypted timestamp     */
    unsigned int adv_winverf;     /* encrypted window verifier */
};

/*
* Verifier: server variety
* The server returns (encrypted) the same timestamp the client
* gave it minus one second. It also tells the client its nickname
* to be used in future transactions (unencrypted).
*/
struct authdes_verf_svr {
timestamp adv_timeverf;       /* encrypted verifier    */
unsigned int adv_nickname;    /* new nickname for client */
};
```

Diffie-Hellman Encryption

In this scheme, there are two constants, `BASE` and `MODULUS`. The particular values Sun has chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS = "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b
```

The way this scheme works is best explained by an example. Suppose there are two people "A" and "B" who want to send encrypted messages to each other. So, A and B both generate "secret" keys at random which they do not reveal to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their "public" keys. These keys are computed as follows:

```
PK(A) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

The "**" notation is used here to represent exponentiation. Now, both A and B can arrive at the "common" key between them, represented here as CK(A, B), without revealing their secret keys.

A computes:

```
CK(A, B) = ( PK(B) ** SK(A)) mod MODULUS
```

while B computes:

```
CK(A, B) = ( PK(A) ** SK(B)) mod MODULUS
```

These two can be shown to be equivalent:

```
(PK(B) ** SK(A)) mod MODULUS = (PK(A) ** SK(B)) mod MODULUS
```

We drop the "mod MODULUS" parts and assume modulo arithmetic to simplify things:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then, replace PK(B) by what B computed earlier and likewise for PK(A).

```
((BASE ** SK(B)) ** SK(A) = (BASE ** SK(A)) ** SK(B)
```

which leads to:

```
BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) * SK(B))
```

This common key CK(A, B) is not used to encrypt the timestamps used in the protocol. Rather, it is used only to encrypt a conversation key which is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, since conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8-bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

7.6. Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). Sun uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to (2**31) - 1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values—a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit

of the header; the length is the 31 low-order bits. (Note that this record specification is NOT in XDR standard form!)

7.7. The RPC Language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language.

An Example Service Described in the RPC Language

Here is an example of the specification of a simple ping program.

```
/*
* Simple ping program
*/
program PING_PROG {
     /* Latest and greatest version */
     version PING_VERS_PINGBACK {
     void
     PINGPROC_NULL(void) = 0;

     /*
     * Ping the caller, return the round-trip time
     * (in microseconds). Returns -1 if the operation
     * timed out.
     */
     int
     PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
* Original version
*/
version PING_VERS_ORIG {
     void
     PINGPROC_NULL(void) = 0;
     } = 1;
} = 1;

const PING_VERS = 2;        /* latest version */
```

The first version described is `PING_VERS_PINGBACK` with two procedures, `PINGPROC_NULL` and `PINGPROC_PINGBACK`. `PINGPROC_NULL` takes no arguments and returns no results, but it is useful for computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the same semantics, and never require any kind of authentication. The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it returns the amount of time (in microseconds) that the operation used. The next version, `PING_VERS_ORIG`, is the original version of the protocol and it does not contain `PINGPROC_PINGBACK` procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

The RPC Language Specification

The RPC language is identical to the XDR language, except for the added definition of a `program-def` described below.

```
program-def:
    "program" identifier "{"
        version-def
        version-def *
    "}" "=" constant ";"

version-def:
    "version" identifier "{"
        procedure-def
        procedure-def *
    "}" "=" constant ";"

procedure-def:
    type-specifier identifier "(" type-specifier ")"
    "=" constant ";"
```

Syntax Notes

1. The following keywords are added and cannot be used as identifiers: "program" and "version";
2. A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
3. A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.
4. Program identifiers are in the same name space as constant and type identifiers.
5. Only unsigned constants can be assigned to programs, versions and procedures.

7.8. Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;          /* portmapper port number */

/*
* A mapping of (program, version, protocol) to port number
*/
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};

/*
* Supported values for the "prot" field
*/
const IPPROTO_TCP = 6;          /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;         /* protocol number for UDP/IP */

/*
* A list of mappings
*/
struct *pmaplist {
    mapping map;
    pmaplist next;
};

/*
* Arguments to callit
*/
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};

/*
* Results of callit
*/
struct call_result {
    unsigned int port;
    opaque res<>;
};

/*
* Port mapper procedures
*/
program PMAP_PROG {
    version PMAP_VERS {
        void
        PMAPPROC_NULL(void)          = 0;

        bool
```

```
            PMAPPROC_SET(mapping)        = 1;

            bool
            PMAPPROC_UNSET(mapping)      = 2;

            unsigned int
            PMAPPROC_GETPORT(mapping)    = 3;

            pmaplist
            PMAPPROC_DUMP(void)          = 4;

            call_result
            PMAPPROC_CALLIT(call_args)   = 5;
        } = 2;
    } = 100000;
```

Port Mapper Operation

The portmapper program currently supports two protocols (UDP/IP and TCP/IP). The portmapper is contacted by talking to it on assigned port number 111 (SUNRPC [8]) on either of these protocols. The following is a description of each of the portmapper procedures:

PMAPPROC_NULL:
This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

PMAPPROC_SET:
When a program first becomes available on a machine, it registers itself with the port mapper program on the same machine. The program passes its program number "prog", version number "vers", transport protocol number "prot", and the port "port" on which it awaits service request. The procedure returns a boolean response whose value is `TRUE` if the procedure successfully established the mapping and `FALSE` otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(prog, vers, prot)".

PMAPPROC_UNSET:
When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of `PMAPPROC_SET`. The protocol and port number fields of the argument are ignored.

PMAPPROC_GETPORT:
Given a program number "prog", version number "vers", and transport protocol number "prot", this procedure returns the port number on which the program is awaiting call requests. A port value of zeros means the program has not been registered. The "port" field of the argument is ignored.

PMAPPROC_DUMP:
This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT:
This procedure allows a caller to call another remote procedure on the same

machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters "prog", "vers", "proc", and the bytes of "args" are the program number, version number, procedure number, and parameters of the remote procedure. Note:

1. This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.
2. The port mapper communicates with the remote program using UDP/IP only.

The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

7.9. References

[1] Birrell, Andrew D. & Nelson, Bruce Jay; "Implementing Remote Procedure Calls"; XEROX CSL-83-7, October 1983.

[2] Cheriton, D.; "VMTP: Versatile Message Transaction Protocol", Preliminary Version 0.3; Stanford University, January 1987.

[3] Diffie & Hellman; "New Directions in Cryptography"; IEEE Transactions on Information Theory IT-22, November 1976.

[4] Harrenstien, K.; "Time Server", RFC 738; Information Sciences Institute, October 1977.

[5] National Bureau of Standards; "Data Encryption Standard"; Federal Information Processing Standards Publication 46, January 1977.

[6] Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793; Information Sciences Institute, September 1981.

[7] Postel, J.; "User Datagram Protocol", RFC 768; Information Sciences Institute, August 1980.

[8] Reynolds, J. & Postel, J.; "Assigned Numbers", RFC 923; Information Sciences Institute, October 1984.

8

Network File System: Version 2 Protocol Specification

8.1. Status of this Standard

Note: This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It specifies it in standard ARPA RFC form.

8.2. Introduction

The Sun Network Filesystem (NFS) protocol provides transparent remote access to shared filesystems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an External Data Representation (XDR). Implementations exist for a variety of machines, from personal computers to supercomputers.

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. It performs the operating system-specific functions that allow, for example, to attach remote directory trees to some local file system.

Remote Procedure Call

Sun's remote procedure call specification provides a procedure- oriented interface to remote services. Each server supplies a program that is a set of procedures. NFS is one such "program". The combination of host address, program number, and procedure number specifies one remote service procedure. RPC does not depend on services provided by specific protocols, so it can be used with any underlying transport protocol. See the *Remote Procedure Calls: Protocol Specification* chapter of this manual.

External Data Representation

The External Data Representation (XDR) standard provides a common way of representing a set of data types over a network. The NFS Protocol Specification is written using the RPC data description language. For more information, see the *External Data Representation Standard: Protocol Specification* chapter of this manual. Sun provides implementations of XDR and RPC, but NFS does not require their use. Any software that provides equivalent functionality can be used, and if the encoding is exactly the same it can interoperate with other implementations of NFS.

Stateless Servers

The NFS protocol is stateless. That is, a server does not need to maintain any extra state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a failure. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed, or the network temporarily went down. The client of a stateful server, on the other hand, needs to either detect a server crash and rebuild the server's state when it comes back up, or cause client operations to fail.

This may not sound like an important issue, but it affects the protocol in some unexpected ways. We feel that it is worth a bit of extra complexity in the protocol to be able to write very simple servers that do not require fancy crash recovery.

On the other hand, NFS deals with objects such as files and directories that inherently have state -- what good would a file be if it did not keep its contents intact? The goal is to not introduce any extra state in the protocol itself. Another way to simplify recovery is by making operations "idempotent" whenever possible (so that they can potentially be repeated).

8.3. NFS Protocol Definition

Servers have been known to change over time, and so can the protocol that they use. So RPC provides a version number with each RPC request. This RFC describes version two of the NFS protocol. Even in the second version, there are various obsolete procedures and parameters, which will be removed in later versions. An RFC for version three of the NFS protocol is currently under preparation.

File System Model

NFS assumes a file system that is hierarchical, with directories as all but the bottom-level files. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used, as well as using different syntax to represent the "pathname", which is the concatenation of all the "components" (directory and file names) in the name. A "file system" is a tree on a single server (usually a single disk or physical partition) with a specified "root". Some operating systems provide a "mount" operation to make all file systems appear as a single tree, while others maintain a "forest" of file systems. Files are unstructured streams of uninterpreted bytes. Version 3 of NFS uses a slightly more general file system model.

NFS looks up one component of a pathname at a time. It may not be obvious why it does not just take the whole pathname, traipse down the directories, and return a file handle when it is done. There are several good reasons not to do this. First, pathnames need separators between the directory components, and different operating systems use different separators. We could define a Network Standard Pathname Representation, but then every pathname would have to be parsed and converted at each end. Other issues are discussed in *NFS Implementation Issues* below.

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This provides a network standard format for representing directories. The same argument as above could have

been used to justify a procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would be just too slow.

RPC Information

Authentication

The NFS service uses `AUTH_UNIX`, `AUTH_DES`, or `AUTH_SHORT` style authentication, except in the NULL procedure where `AUTH_NONE` is also allowed.

Transport Protocols

NFS currently is supported on UDP/IP only.

Port Number

The NFS protocol currently uses the UDP port number 2049. This is not an officially assigned port, so later versions of the protocol use the "Portmapping" facility of RPC.

Sizes of XDR Structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/* The maximum number of bytes of data in a READ or WRITE request */
const MAXDATA = 8192;

/* The maximum number of bytes in a pathname argument */
const MAXPATHLEN = 1024;

/* The maximum number of bytes in a file name argument */
const MAXNAMLEN = 255;

/* The size in bytes of the opaque "cookie" passed by READDIR */
const COOKIESIZE = 4;

/* The size in bytes of the opaque file handle */
const FHSIZE = 32;
```

Basic Data Types

The following XDR definitions are basic structures and types used in other structures described further on.

stat

```
enum stat {
    NFS_OK = 0,
    NFSERR_PERM=1,
    NFSERR_NOENT=2,
    NFSERR_IO=5,
    NFSERR_NXIO=6,
    NFSERR_ACCES=13,
    NFSERR_EXIST=17,
    NFSERR_NODEV=19,
    NFSERR_NOTDIR=20,
    NFSERR_ISDIR=21,
    NFSERR_FBIG=27,
    NFSERR_NOSPC=28,
    NFSERR_ROFS=30,
    NFSERR_NAMETOOLONG=63,
    NFSERR_NOTEMPTY=66,
    NFSERR_DQUOT=69,
    NFSERR_STALE=70,
    NFSERR_WFLUSH=99
};
```

The `stat()` type is returned with every procedure's results. A value of `NFS_OK` indicates that the call completed successfully and the results are valid. The other values indicate some kind of error occurred on the server side during the servicing of the procedure. The error values are derived from UNIX error numbers.

NFSERR_PERM:
Not owner. The caller does not have correct ownership to perform the requested operation.

NFSERR_NOENT:
No such file or directory. The file or directory specified does not exist.

NFSERR_IO:
Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFSERR_NXIO:
No such device or address.

NFSERR_ACCES:
Permission denied. The caller does not have the correct permission to perform the requested operation.

NFSERR_EXIST:
File exists. The file specified already exists.

NFSERR_NODEV:
No such device.

NFSERR_NOTDIR:
Not a directory. The caller specified a non-directory in a directory operation.

NFSERR_ISDIR:
Is a directory. The caller specified a directory in a non- directory operation.

NFSERR_FBIG:
File too large. The operation caused a file to grow beyond the server's limit.

NFSERR_NOSPC:
No space left on device. The operation caused the server's filesystem to reach its limit.

NFSERR_ROFS:
Read-only filesystem. Write attempted on a read-only filesystem.

NFSERR_NAMETOOLONG:
File name too long. The file name in an operation was too long.

NFSERR_NOTEMPTY:
Directory not empty. Attempted to remove a directory that was not empty.

NFSERR_DQUOT:
Disk quota exceeded. The client's disk quota on the server has been exceeded.

NFSERR_STALE:
The "fhandle" given in the arguments was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.

NFSERR_WFLUSH:
The server's write cache used in the `WRITECACHE` call got flushed to disk.

`ftype`

```
enum ftype {
    NFNON = 0,
    NFREG = 1,
    NFDIR = 2,
    NFBLK = 3,
    NFCHR = 4,
    NFLNK = 5
};
```

The enumeration `ftype` gives the type of a file. The type `NFNON` indicates a non-file, `NFREG` is a regular file, `NFDIR` is a directory, `NFBLK` is a block-special device, `NFCHR` is a character-special device, and `NFLNK` is a symbolic link.

`fhandle`

```
typedef opaque fhandle[FHSIZE];
```

The `fhandle` is the file handle passed between the server and the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

`timeval`

```
struct timeval {
    unsigned int seconds;
    unsigned int useconds;
};
```

The `timeval` structure is the number of seconds and microseconds since midnight January 1, 1970, Greenwich Mean Time. It is used to pass time and date information.

`fattr`

```
struct fattr {
    ftype        type;
    unsigned int mode;
    unsigned int nlink;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    unsigned int blocksize;
    unsigned int rdev;
    unsigned int blocks;
    unsigned int fsid;
    unsigned int fileid;
    timeval      atime;
    timeval      mtime;
    timeval      ctime;
};
```

The `fattr` structure contains the attributes of a file; "type" is the type of the file; "nlink" is the number of hard links to the file (the number of different names for the same file); "uid" is the user identification number of the owner of the file; "gid" is the group identification number of the group of the file; "size" is the size in bytes of the file; "blocksize" is the size in bytes of a block of the file; "rdev" is the device number of the file if it is type `NFCHR` or `NFBLK`; "blocks" is the number of blocks the file takes up on disk; "fsid" is the file system identifier for the filesystem containing the file; "fileid" is a number that uniquely identifies the file within its filesystem; "atime" is the time when the file was last accessed for either read or write; "mtime" is the time when the file data was last modified (written); and "ctime" is the time when the status of the file was last changed. Writing to the file also changes "ctime" if the size of the file changes.

"mode" is the access mode encoded as a set of bits. Notice that the file type is specified both in the mode bits and in the file type. This is really a bug in the protocol and will be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

Bit	*Description*
0040000	This is a directory; "type" field should be NFDIR.
0020000	This is a character special file; "type" field should be NFCHR.
0060000	This is a block special file; "type" field should be NFBLK.
0100000	This is a regular file; "type" field should be NFREG.
0120000	This is a symbolic link file; "type" field should be NFLNK.
0140000	This is a named socket; "type" field should be NFNON.
0004000	Set user id on execution.
0002000	Set group id on execution.
0001000	Save swapped text even after use.
0000400	Read permission for owner.
0000200	Write permission for owner.
0000100	Execute and search permission for owner.
0000040	Read permission for group.
0000020	Write permission for group.
0000010	Execute and search permission for group.
0000004	Read permission for others.
0000002	Write permission for others.
0000001	Execute and search permission for others.

Notes:

The bits are the same as the mode bits returned by the `stat(2)` system call in the UNIX system. The file type is specified both in the mode bits and in the file type. This is fixed in future versions.

The "rdev" field in the attributes structure is an operating system specific device specifier. It will be removed and generalized in the next revision of the protocol.

`sattr`

```
struct sattr {
    unsigned int mode;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    timeval      atime;
    timeval      mtime;
};
```

The `sattr` structure contains the file attributes which can be set from the client. The fields are the same as for `fattr` above. A "size" of zero means the file should be truncated. A value of -1 indicates a field that should be ignored.

`filename`

```
typedef string filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or pathname components.

`path`

```
typedef string path<MAXPATHLEN>;
```

The type `path` is a pathname. The server considers it as a string with no internal structure, but to the client it is the name of a node in a filesystem tree.

attrstat

```
union attrstat switch (stat status) {
    case NFS_OK:
        fattr attributes;
    default:
        void;
};
```

The `attrstat` structure is a common procedure result. It contains a "status" and, if the call succeeded, it also contains the attributes of the file on which the operation was done.

diropargs

```
struct diropargs {
    fhandle  dir;
    filename name;
};
```

The `diropargs` structure is used in directory operations. The "fhandle" "dir" is the directory in which to find the file "name". A directory operation is one in which the directory is affected.

diropres

```
union diropres switch (stat status) {
    case NFS_OK:
        struct {
            fhandle file;
            fattr   attributes;
        } diropok;
    default:
        void;
};
```

The results of a directory operation are returned in a `diropres` structure. If the call succeeded, a new file handle "file" and the "attributes" associated with that file are returned along with the "status".

Server Procedures

The protocol definition is given as a set of procedures with arguments and results defined using the RPC language. A brief description of the function of each procedure should provide enough information to allow implementation.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. For example, a client `WRITE` request may cause the server to update data blocks, filesystem information blocks (such as indirect blocks), and file attribute information (size and modify times). When the `WRITE` returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests, the client would have to save those requests so that it could resend them in case of a server crash.

```
/*
* Remote file service routines
*/
program NFS_PROGRAM {
    version NFS_VERSION {
        void NFSPROC_NULL(void) = 0;
        attrstat NFSPROC_GETATTR(fhandle)= 1;
        attrstat NFSPROC_SETATTR(sattrargs) = 2;
        void NFSPROC_ROOT(void) = 3;
        diropres NFSPROC_LOOKUP(diropargs)  = 4;
        readlinkres NFSPROC_READLINK(fhandle) = 5;
        readres NFSPROC_READ(readargs) = 6;
        void NFSPROC_WRITECACHE(void) = 7;
        attrstat NFSPROC_WRITE(writeargs) = 8;
        diropres NFSPROC_CREATE(createargs) = 9;
        stat NFSPROC_REMOVE(diropargs) = 10;
        stat NFSPROC_RENAME(renameargs) = 11;
        stat NFSPROC_LINK(linkargs) = 12;
        stat NFSPROC_SYMLINK(symlinkargs) = 13;
        diropres NFSPROC_MKDIR(createargs)  = 14;
        stat NFSPROC_RMDIR(diropargs)   = 15;
        readdirres NFSPROC_READDIR(readdirargs) = 16;
        statfsres NFSPROC_STATFS(fhandle) = 17;
    } = 2;
} = 100003;
```

Do Nothing

```
void
NFSPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Get File Attributes

```
attrstat
NFSPROC_GETATTR (fhandle) = 1;
```

If the reply status is NFS_OK, then the reply attributes contains the attributes for the file given by the input fhandle.

Set File Attributes

```
struct sattrargs {
    fhandle file;
    sattr attributes;
    };

attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

The "attributes" argument contains fields which are either -1 or are the new value for the attributes of "file". If the reply status is NFS_OK, then the reply attributes have the attributes of the file after the "SETATTR" operation has completed.

Note: The use of -1 to indicate an unused field in "attributes" is changed in the next version of the protocol.

Get Filesystem Root

```
void
NFSPROC_ROOT(void) = 3;
```

Obsolete. This procedure is no longer used because finding the root file handle of a filesystem requires moving pathnames between client and server. To do this right we would have to define a network standard representation of pathnames. Instead, the function of looking up the root file handle is done by the `MNTPROC_MNT()` procedure. (See the *Mount Protocol Definition* below for details).

Look Up File Name

```
diropres
NFSPROC_LOOKUP(dirop args) = 4;
```

If the reply "status" is `NFS_OK`, then the reply "file" and reply "attributes" are the file handle and attributes for the file "name" in the directory given by "dir" in the argument.

Read From Symbolic Link

```
union readlinkres switch (stat status) {
    case NFS_OK:
        path data;
    default:
        void;
};

readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

If "status" has the value `NFS_OK`, then the reply "data" is the data in the symbolic link given by the file referred to by the fhandle argument.

Note: since NFS always parses pathnames on the client, the pathname in a symbolic link may mean something different (or be meaningless) on a different client or on the server if a different pathname syntax is used.

Read From File

```
struct readargs {
    fhandle file;
    unsigned offset;
    unsigned count;
    unsigned totalcount;
};

union readres switch (stat status) {
    case NFS_OK:
        fattr attributes;
        opaque data<NFS_MAXDATA>;
    default:
        void;
};

readres
NFSPROC_READ(readargs) = 6;
```

Returns up to "count" bytes of "data" from the file given by "file", starting at

"offset" bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in "attributes".

Note: The argument "totalcount" is unused, and is removed in the next protocol revision.

Write to Cache

```
void
NFSPROC_WRITECACHE(void) = 7;
```

To be used in the next protocol revision.

Write to File

```
struct writeargs {
    fhandle file;
    unsigned beginoffset;
    unsigned offset;
    unsigned totalcount;
    opaque data<NFS_MAXDATA>;
};

attrstat
NFSPROC_WRITE(writeargs) = 8;
```

Writes "data" beginning "offset" bytes from the beginning of "file". The first byte of the file is at offset zero. If the reply "status" is NFS_OK, then the reply "attributes" contains the attributes of the file after the write has completed. The write operation is atomic. Data from this call to `WRITE` will not be mixed with data from another client's calls.

Note: The arguments "beginoffset" and "totalcount" are ignored and are removed in the next protocol revision.

Create File

```
struct createargs {
    diropargs where;
    sattr attributes;
};

diropres
NFSPROC_CREATE(createargs) = 9;
```

The file "name" is created in the directory given by "dir". The initial attributes of the new file are given by "attributes". A reply "status" of NFS_OK indicates that the file was created, and reply "file" and reply "attributes" are its file handle and attributes. Any other reply "status" means that the operation failed and no file was created.

Note: This routine should pass an exclusive create flag, meaning "create the file only if it is not already there".

Remove File

```
stat
NFSPROC_REMOVE(diropargs) = 10;
```

The file "name" is removed from the directory given by "dir". A reply of NFS_OK means the directory entry was removed.

Note: possibly non-idempotent operation.

Rename File

```
struct renameargs {
    diropargs from;
    diropargs to;
};

stat
NFSPROC_RENAME(renameargs) = 11;
```

The existing file "from.name" in the directory given by "from.dir" is renamed to "to.name" in the directory given by "to.dir". If the reply is NFS_OK, the file was renamed. The RENAME operation is atomic on the server; it cannot be interrupted in the middle.

Note: possibly non-idempotent operation.

Create Link to File

```
struct linkargs {
    fhandle from;
    diropargs to;
};

stat
NFSPROC_LINK(linkargs) = 12;
```

Creates the file "to.name" in the directory given by "to.dir", which is a hard link to the existing file given by "from". If the return value is NFS_OK, a link was created. Any other return value indicates an error, and the link was not created.

A hard link should have the property that changes to either of the linked files are reflected in both files. When a hard link is made to a file, the attributes for the file should have a value for "nlink" that is one greater than the value before the link.

Note: possibly non-idempotent operation.

Create Symbolic Link

```
struct symlinkargs {
    diropargs from;
    path to;
    sattr attributes;
};

stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

Creates the file "from.name" with ftype NFLNK in the directory given by "from.dir". The new file contains the pathname "to" and has initial attributes given by "attributes". If the return value is NFS_OK, a link was created. Any other return value indicates an error, and the link was not created.

A symbolic link is a pointer to another file. The name given in "to" is not interpreted by the server, only stored in the newly created file. When the client references a file that is a symbolic link, the contents of the symbolic link are normally transparently reinterpreted as a pathname to substitute. A READLINK operation

returns the data to the client for interpretation.

Note: On UNIX servers the attributes are never used, since symbolic links always have mode 0777.

Create Directory

```
diropres
NFSPROC_MKDIR (createargs) = 14;
```

The new directory "where.name" is created in the directory given by "where.dir". The initial attributes of the new directory are given by "attributes". A reply "status" of NFS_OK indicates that the new directory was created, and reply "file" and reply "attributes" are its file handle and attributes. Any other reply "status" means that the operation failed and no directory was created.

Note: possibly non-idempotent operation.

Remove Directory

```
stat
NFSPROC_RMDIR(diropargs) = 15;
```

The existing empty directory "name" in the directory given by "dir" is removed. If the reply is NFS_OK, the directory was removed.

Note: possibly non-idempotent operation.

Read From Directory

```
struct readdirargs {
    fhandle dir;
    nfscookie cookie;
    unsigned count;
};

struct entry {
    unsigned fileid;
    filename name;
    nfscookie cookie;
    entry *nextentry;
};

union readdirres switch (stat status) {
    case NFS_OK:
        struct {
            entry *entries;
            bool eof;
        } readdirok;
    default:
        void;
};

readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

Returns a variable number of directory entries, with a total size of up to "count" bytes, from the directory given by "dir". If the returned value of "status" is NFS_OK, then it is followed by a variable number of "entry"s. Each "entry" contains a "fileid" which consists of a unique number to identify the file within a

filesystem, the "name" of the file, and a "cookie" which is an opaque pointer to the next entry in the directory. The cookie is used in the next READDIR call to get more entries starting at a given point in the directory. The special cookie zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The "fileid" field should be the same number as the "fileid" in the attributes of the file. (See the *Basic Data Types* section.) The "eof" flag has a value of TRUE if there are no more entries in the directory.

Get Filesystem Attributes

```
union statfsres (stat status) {
    case NFS_OK:
        struct {
            unsigned tsize;
            unsigned bsize;
            unsigned blocks;
            unsigned bfree;
            unsigned bavail;
        } info;
    default:
        void;
};

statfsres
NFSPROC_STATFS(fhandle) = 17;
```

If the reply "status" is NFS_OK, then the reply "info" gives the attributes for the filesystem that contains file referred to by the input fhandle. The attribute fields contain the following values:

tsize:
The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of READ and WRITE requests.

bsize:
The block size in bytes of the filesystem.

blocks:
The total number of "bsize" blocks on the filesystem.

bfree:
The number of free "bsize" blocks on the filesystem.

bavail:
The number of "bsize" blocks available to non-privileged users.

Note: This call does not work well if a filesystem has variable size blocks.

8.4. NFS Implementation Issues

The NFS protocol is designed to be operating system independent, but since this version was designed in a UNIX environment, many operations have semantics similar to the operations of the UNIX file system. This section discusses some of the implementation-specific semantic issues.

Server/Client Relationship

The NFS protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client wants to implement complicated filesystem semantics.

For example, some operating systems allow removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the filesystem. It is impossible for a stateless server to implement these semantics. The client can do some tricks such as renaming the file on remove, and only removing it on close. We believe that the server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also potentially be a server, and remote and local mounted filesystems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote filesystem and reaches the mount point on the server for another remote filesystem. Allowing the server to follow the second remote mount would require loop detection, server lookup, and user revalidation. Instead, we decided not to let clients cross a server's mount point. When a client does a LOOKUP on a directory on which the server has mounted a filesystem, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

Pathname Interpretation

There are a few complications to the rule that pathnames are always parsed on the client. For example, symbolic links could have different interpretations on different clients. Another common problem for non-UNIX implementations is the special interpretation of the pathname ".." to mean the parent of a given directory. The next revision of the protocol uses an explicit flag to indicate the parent instead.

Permission Issues

The NFS protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal operating system permission checking using `AUTH_UNIX` style authentication as the basis of its protection mechanism. The server gets the client's effective "uid", effective "gid", and groups on each call and uses them to check permission. There are various problems with this method that can been resolved in interesting ways.

Using "uid" and "gid" implies that the client and server share the same "uid" list. Every server and client pair must have the same mapping from user to "uid" and from group to "gid". Since every client can also be a server, this tends to imply that the whole network shares the same "uid/gid" space. `AUTH_DES` (and the next revision of the NFS protocol) uses string names instead of numbers, but there are still complex problems to be solved.

Another problem arises due to the usually stateful open operation. Most operating systems check permission at open time, and then check that the file is open on each read and write request. With stateless servers, the server has no idea that the file is open and must do permission checking on each read and write call. On a local filesystem, a user can open a file and then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is

open. On a remote filesystem, by contrast, the write would fail. To get around this problem, the server's permission checking algorithm should allow the owner of a file to access it regardless of the permission setting.

A similar problem has to do with paging in from a file over the network. The operating system usually checks for execute permission before opening a file for demand paging, and then reads blocks from the open file. The file may not have read permission, but after it is opened it doesn't matter. An NFS server can not tell the difference between a normal file read and a demand page-in read. To make this work, the server allows reading of files if the "uid" given in the call has execute or read permission on the file.

In most operating systems, a particular user (on the user ID zero) has access to all files no matter what permission and ownership they have. This "super-user" permission may not be allowed on the server, since anyone who can become super-user on their workstation could gain access to all remote files. The UNIX server by default maps user id 0 to -2 before doing its access checking. This works except for NFS root filesystems, where super-user access cannot be avoided.

Setting RPC Parameters

Various file system parameters and options should be set at mount time. The mount protocol is described in the appendix below. For example, "Soft" mounts as well as "Hard" mounts are usually both provided. Soft mounted file systems return errors when RPC operations fail (after a given number of optional retransmissions), while hard mounted file systems continue to retransmit forever. Clients and servers may need to keep caches of recent operations to help avoid problems with non-idempotent operations.

8.5. Mount Protocol Definition

Introduction

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system specific services to get the NFS off the ground -- looking up server path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only, for example, to warn possible clients when a server is going down.

Version one of the mount protocol is used with version two of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

RPC Information

Authentication
The mount service uses `AUTH_UNIX` and `AUTH_DES` style authentication only.

Transport Protocols
The mount service is currently supported on UDP/IP only.

Port Number
Consult the server's portmapper, described in the *Remote Procedure Calls: Protocol Specification*, to find the port number on which the mount service is registered.

Sizes of XDR Structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/* The maximum number of bytes in a pathname argument */
const MNTPATHLEN = 1024;

/* The maximum number of bytes in a name argument */
const MNTNAMLEN = 255;

/* The size in bytes of the opaque file handle */
const FHSIZE = 32;
```

Basic Data Types

This section presents the data types used by the mount protocol. In many cases they are similar to the types used in NFS.

`fhandle`

```
typedef opaque fhandle[FHSIZE];
```

The type `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the "fhandle" XDR definition in version 2 of the NFS protocol; see *Basic Data Types* in the definition of the NFS protocol, above.

`fhstatus`

```
union fhstatus switch (unsigned status) {
    case 0:
        fhandle directory;
    default:
        void;
};
```

The type `fhstatus` is a union. If a "status" of zero is returned, the call completed successfully, and a file handle for the "directory" follows. A non-zero status indicates some sort of error. In this case the status is a UNIX error number.

`dirpath`

```
typedef string dirpath<MNTPATHLEN>;
```

The type `dirpath` is a server pathname of a directory.

name

```
typedef string name<MNTNAMLEN>;
```

The type name is an arbitrary string used for various names.

Server Procedures

The following sections define the RPC procedures supplied by a mount server.

```
/*
* Protocol description for the mount program
*/

program MOUNTPROG {
/*
* Version 1 of the mount protocol used with
* version 2 of the NFS protocol.
*/
    version MOUNTVERS {
        void MOUNTPROC_NULL(void) = 0;
        fhstatus MOUNTPROC_MNT(dirpath) = 1;
        mountlist MOUNTPROC_DUMP(void) = 2;
        void MOUNTPROC_UMNT(dirpath) = 3;
        void MOUNTPROC_UMNTALL(void) = 4;
        exportlist MOUNTPROC_EXPORT(void) = 5;
    } = 1;
} = 100005;
```

Do Nothing

```
void
MNTPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Add Mount Entry

```
fhstatus
MNTPROC_MNT(dirpath) = 1;
```

If the reply "status" is 0, then the reply "directory" contains the file handle for the directory "dirname". This file handle may be used in the NFS protocol. This procedure also adds a new entry to the mount list for this client mounting "dirname".

Return Mount Entries

```
struct *mountlist {
    name      hostname;
    dirpath   directory;
    mountlist nextentry;
};

mountlist
MNTPROC_DUMP(void) = 2;
```

Returns the list of remote mounted filesystems. The "mountlist" contains one entry for each "hostname" and "directory" pair.

Remove Mount Entry

```
void
MNTPROC_UMNT(dirpath) = 3;
```

Removes the mount list entry for the input "dirpath".

Remove All Mount Entries

```
void
MNTPROC_UMNTALL(void) = 4;
```

Removes all of the mount list entries for this client.

Return Export List

```
struct *groups {
    name grname;
    groups grnext;
};

struct *exportlist {
    dirpath filesys;
    groups groups;
    exportlist next;
};

exportlist
MNTPROC_EXPORT(void) = 5;
```

Returns a variable number of export list entries. Each entry contains a filesystem name and a list of groups that are allowed to import it. The filesystem name is in "filesys", and the group name is in the list "groups".

Note: The exportlist should contain more information about the status of the filesystem, such as a read-only flag.

PART THREE: Transport-Level Programming

9

Transport Level Interface Programming

This chapter provides detailed information, with various examples, on the UNIX system Transport Interface. This interface is intended to supercede the socket-based interprocess communications mechanisms as the standard means of gaining direct access to transport services. Network application developers who do *not* require such direct access should instead work within the Remote Procedure Call (RPC) framework — which is documented in PART I of this manual.

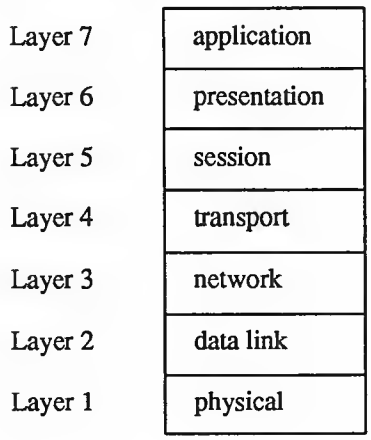
NOTE *SunOS 4.1 does not support RPC on TLI. This is a feature that will appear in future products.*

The following discussion assumes a working knowledge of UNIX system programming and data communication concepts. Familiarity with the Reference Model of Open Systems Interconnection (OSI) is required as well.

9.1. Background

To place the Transport Interface in perspective, a discussion of the OSI Reference Model is first presented. The Reference Model partitions networking functions into seven layers, as depicted in Figure 9-1.

Figure 9-1 *OSI Reference Model*



Layer 1
: The physical layer is responsible for the transmission of raw data over a communication medium.

Layer 2
: The data link layer provides the exchange of data between network layer entities. It detects and corrects any errors that may occur in the physical layer transmission.

Layer 3
: The network layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layer entities within the network.

Layer 4
: The transport layer provides transparent data transfer services between session layer entities by relieving them from concerns of how reliable and cost-effective transfer of data is achieved.

Layer 5
: The session layer provides the services needed by presentation layer entities that enable them to organize and synchronize their dialogue and manage their data exchange.

Layer 6
: The presentation layer manages the representation of information that application layer entities either communicate or reference in their communication.

Layer 7
: The application layer serves as the window between corresponding application processes that are exchanging information.

A basic principle of the Reference Model is that each layer provides services needed by the next higher layer in a way that frees the upper layer from concern about how these services are provided. This approach simplifies the design of each particular layer.

Industry standards either have been or are being defined at each layer of the Reference Model. Two standards are defined at each layer: one that specifies an interface to the services of the layer, and one that defines the protocol by which services are provided. A service interface standard at any layer frees users of the service from details of how that layer's protocol is implemented, or even which protocol is used to provide the service.

The transport layer is important because it is the lowest layer in the Reference Model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, however, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the International Standards Organization (ISO) protocols, the Transmission Control Protocol and Internet Protocol (TCP/IP) of the ARPANET, Xerox Network Systems (XNS), and the Systems Network Architecture (SNA).

A transport service interface, then, enables applications and higher layer protocols to be implemented without knowledge of the underlying protocol suite.

That is a principle goal of the UNIX system Transport Interface. Also, because an inherent characteristic of the transport layer is that it hides details of the physical medium being used, the Transport Interface offers both protocol and medium independence to networking applications and higher layer protocols.

The UNIX system Transport Interface was modeled after the industry standard ISO Transport Service Definition (ISO 8072). As such, it is intended for those applications and protocols that require transport services. Because the Transport Interface provides reliable data transfer, and because its services are common to several protocol suites, many networking applications will find these services useful.

The Transport Interface is implemented as a user library using the STREAMS input/output mechanism. Therefore, many services available to STREAMS applications are also available to users of the Transport Interface. These services will be highlighted throughout this guide. For detailed information about STREAMS, see the *STREAMS Programming* manual.

9.2. Document Organization

This section is organized as follows:

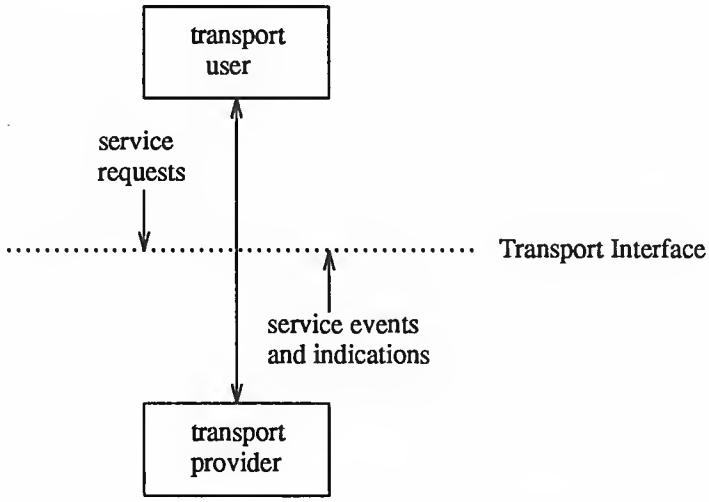
- **Overview of the Transport Interface**, a summary of the basic services available to Transport Interface users and a presentation of the background information needed for the remainder of the section.
- **Introduction to Connection-Mode Service**, a description of the services associated with connection-based (or virtual circuit) communication.
- **Introduction to Connectionless-Mode Service**, a description of the services associated with connectionless (or datagram) communication.
- **A Read/Write Interface**, a description of how users can use the services of `read`(2) and `write`(2) to communicate over a transport connection.
- **Advanced Topics**, a discussion of important concepts not covered in earlier sections. These include asynchronous event handling and processing of multiple, simultaneous connect requests.
- **State Transitions**, an appendix which defines the allowable state transitions associated with the Transport Interface.
- **Guidelines for Protocol Independence**, an appendix which establishes necessary guidelines for developing software that can be run without change over any transport protocol developed for the Transport Interface.
- **Examples**, an appendix that presents the full listing of each programming example used throughout the guide.
- **Glossary**, a definition of the Transport Interface terms and acronyms used in this section.

This section describes the more important and common facilities of the Transport Interface, but is not meant to be exhaustive. Section 3N of the *SunOS Reference Manual* contains a complete description of each Transport Interface routine.

9.3. Overview of the Transport Interface

This section presents a high level overview of the services of the Transport Interface, which supports the transfer of data between two user processes. Figure 9-2 illustrates the Transport Interface.

Figure 9-2 *Transport Interface*



The transport provider is the entity that provides the services of the Transport Interface, and the transport user is the entity that requires these services. An example of a transport provider is the ISO transport protocol, while a transport user may be a networking application or session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

The Network Services Library of UNIX System V includes a set of functions that support the services of the Transport Interface for user processes [see `intro`(3)]. These functions enable a user to initiate requests to the provider and process incoming events. Programs using the Transport Interface can link the appropriate routines as follows:

```
cc prog.c -lnsl_s
```

Modes of Service

Two modes of service, connection-mode and connectionless-mode, are provided by the Transport Interface. Connection-mode is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-mode, in contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple

units. This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained. Connectionless-mode service is attractive for applications that:

- involve short-term request/response interactions
- exhibit a high level of redundancy
- are dynamically reconfigurable
- do not require guaranteed, in-sequence delivery of data

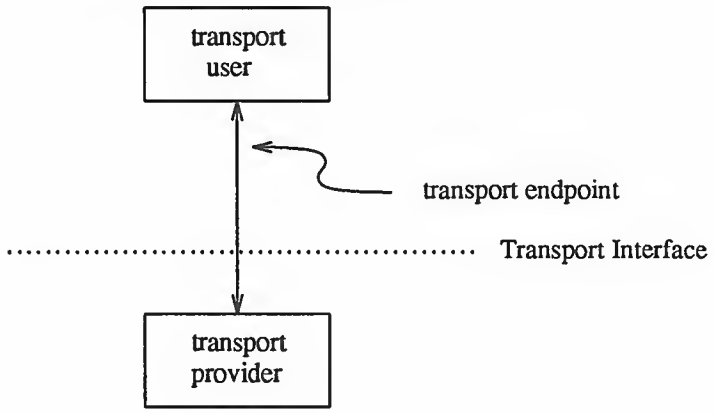
Connection-Mode Service

The connection-mode transport service is characterized by four phases: local management, connection establishment, data transfer, and connection release.

Local Management

The local management phase defines local operations between a transport user and a transport provider. For example, a user must establish a channel of communication with the transport provider, as illustrated in Figure 9-3. Each channel between a transport user and transport provider is a unique endpoint of communication, and will be called the transport endpoint. The `t_open`(3N) routine enables a user to choose a particular transport provider that will supply the connection-mode services, and establishes the transport endpoint.

Figure 9-3 *Channel Between User and Provider*



Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a transport address. More accurately, a transport address is associated with each transport endpoint, and one user process may manage several transport endpoints. In connection-mode service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address may be as simple as a random character string (for example, "file_server"), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport

provider defines its own mechanism for identifying users. Addresses may be assigned to each transport endpoint by `t_bind`(3N).

Table 9-1 *Local Management Routines*

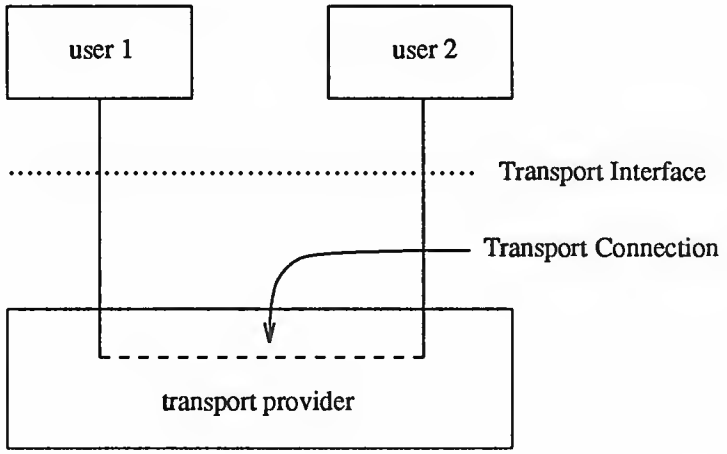
Command	Description
t_alloc	Allocates Transport Interface data structures.
t_bind	Binds a transport address to a transport endpoint.
t_close	Closes a transport endpoint.
t_error	Prints a Transport Interface error message.
t_free	Frees structures allocated using `t_alloc`.
t_getinfo	Returns a set of parameters associated with a particular transport provider.
t_getstate	Returns the state of a transport endpoint.
t_look	Returns the current event on a transport endpoint.
t_open	Establishes a transport endpoint connected to a chosen transport provider.
t_optmgmt	Negotiates protocol-specific options with the transport provider.
t_sync	Synchronizes a transport endpoint with the transport provider.
t_unbind	Unbinds a transport address from a transport endpoint.

In addition to `t_open` and `t_bind`, several routines are available to support local operations. Table 9-1 summarizes all local management routines of the Transport Interface.

Connection Establishment

The connection establishment phase enables two users to create a connection, or virtual circuit, between them, as demonstrated in Figure 9-4.

Figure 9-4 *Transport Connection*



This phase is illustrated by a client-server relationship between two transport users. One user, the server, typically advertises some service to a group of users, and then listens for requests from those users. As each client requires the service, it attempts to connect itself to the server using the server's advertised transport address. The `t_connect`(3N) routine initiates the connect request. One argument to `t_connect`, the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using `t_listen`(3N), and may call `t_accept`(3N) to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Table 9-2 summarizes all routines available for establishing a transport connection.

Table 9-2 *Connection Establishment Routines*

Command	Description
t_accept	Accepts a request for a transport connection.
t_connect	Establishes a connection with the transport user at a specified destination.
t_listen	Retrieves an indication of a connect request from another transport user.
t_rcvconnect	Completes connection establishment if `t_connect` was called in asynchronous mode (see the *Advanced Topics* section).

Data Transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, `t_snd`(3N) and `t_rcv`(3N), send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent. Table 9-3 summarizes the connection mode data transfer routines.

Table 9-3 *Connection Mode Data Transfer Routines*

Command	Description
t_rcv	Retrieves data that has arrived over a transport connection.
t_snd	Send data over an established transport connection.

Connection Release

The connection release phase provides a mechanism for breaking an established connection. When you decide that the conversation should terminate, you can request that the provider release the transport connection. Two types of connection release are supported by the Transport Interface. The first is an abortive release, which directs the transport provider to release the connection immediately. Any previously sent data that has not yet reached the other transport user may be discarded by the transport provider. The `t_snddis`(3N) routine initiates this abortive disconnect, and `t_rcvdis`(3N) processes the incoming indication of an abortive disconnect.

All transport providers must support the abortive release procedure. In addition, some transport providers may also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions `t_sndrel`(3N) and `t_rcvrel`(3N) support this capability. Table 9-4 summarizes the connection release routines.

Table 9-4 *Connection Release Routines*

Command	Description
t_rcvdis	Returns an indication of an aborted connection, including a reason code and user data.
t_rcvrel	Returns an indication that the remote user has requested an orderly release of a connection.
t_snddis	Aborts a connection or rejects a connect request.
t_sndrel	Requests the orderly release of a connection.

Connectionless-Mode Service

The connectionless-mode transport service is characterized by two phases: local management and data transfer. The local management phase defines the same local operations described above for the connection-mode service.

The data transfer phase enables a user to transfer data units (sometimes called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. Two routines, `t_sndudata`(3N) and `t_rcvudata`(3N) support this message-based data transfer facility. Table 9-5 summarizes all routines associated with connectionless-mode data transfer.

Table 9-5 *Connectionless-mode Data Transfer Routines*

Command	Description
t_rcvudata	Retrieves a message sent by another transport user.
t_rcvuderr	Retrieves error information associated with a previously sent message.
t_sndudata	Sends a message to the specified destination user.

State Transitions

The Transport Interface has two components:

- the library routines that provide the transport services to users
- the state transition rules that define the sequence in which the transport routines may be invoked

The state transition rules can be found in the *State Transitions* section of this chapter in the form of state tables. The state tables define the legal sequence of library calls based on state information and the handling of events. These events include user-generated library calls, as well as provider-generated event indications.

NOTE *Any user of the Transport Interface must completely understand all possible state transitions before writing software using the interface.*

9.4. Introduction to Connection-Mode Services

This section describes the connection-mode service of the Transport Interface. As discussed in the previous section, the connection-mode service can be illustrated using a client-server paradigm. The important concepts of connection-mode service will be presented using two programming examples. The examples are related in that the first illustrates how a client establishes a connection to a server and then communicates with the server. The second example shows the server's side of the interaction. All examples discussed in this guide are presented in their entirety in the *Some Examples* section, below.

In the examples, the client establishes a connection with a server process. The server then transfers a file to the client. The client, in turn, receives the data from the server and writes it to its standard output file.

Local Management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using `t_open`, and establish its identity (or address) using `t_bind`.

The set of services supported by the Transport Interface may not be implemented by all transport protocols. Each transport provider has a set of characteristics associated with it that determine the services it offers and the limits associated with those services. This information is returned to the user by `t_open`, and consists of the following:

addr
: maximum size of a transport address

options
: maximum bytes of protocol-specific options that may be passed between the transport user and transport provider

tsdu
: maximum message size that may be transmitted in either connection-mode or connectionless-mode

etsdu
: maximum expedited data message size that may be sent over a transport connection

connect
: maximum number of bytes of user data that may be passed between users during connection establishment

discon
: maximum bytes of user data that may be passed between users during the abortive release of a connection

servtype
: the type of service supported by the transport provider

The three service types defined by the Transport Interface are:

T_COTS
: The transport provider supports connection-mode service but does not provide the optional orderly release facility.

T_COTS_ORD
: The transport provider supports connection-mode service with the optional orderly release facility.

T_CLTS
: The transport provider supports connectionless-mode service. Only one such service can be associated with the transport provider identified by `t_open`.

NOTE `t_open` returns the default provider characteristics associated with a transport endpoint. However, some characteristics may change after an endpoint has been opened. This will occur if the characteristics are associated with negotiated options (option negotiation is described later in this section). For example, if the

support of expedited data transfer is a negotiated option, the value of this characteristic may change. `t_getinfo` may be called to retrieve the current characteristics of a transport endpoint.

Once a user establishes a transport endpoint with the chosen transport provider, it must establish its identity. As mentioned earlier, `t_bind` accomplishes this by binding a transport address to the transport endpoint. In addition, for servers, this routine informs the transport provider that the endpoint will be used to listen for incoming connect requests, also called connect indications.

An optional facility, `t_optmgmt`(3N), is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol is expected to define its own set of negotiable protocol options, which may include such information as Quality-of-Service parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The Client

The local management requirements of the example client and server are used to discuss details of these facilities. The following are the definitions needed by the client program, followed by its necessary local management steps.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR    1    /* server's well known address */

main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }
```

The first argument to `t_open` is the pathname of a file system node that identifies the transport protocol that will supply the transport service. In this example, `/dev/tivc` is a STREAMS `clone` device node that identifies a generic, connection-based transport protocol [see `clone`(4)].

NOTE *The name* `/dev/tivc` *does not exist in SunOS. This is just a name used as an example that represents the transport selection node.*

The `clone` device finds an available minor device of the transport provider for the user. It is opened for both reading and writing, as specified by the O_RDWR open flag. The third argument may be used to return the service characteristics of the transport provider to the user. This information is useful when writing protocol-independent software (discussed in the *Guidelines for Protocol Independence* section, below.) For simplicity, the client and server in this example ignore this information and assume the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the T_COTS_ORD service type, and the example will use the orderly release facility to release the connection.
- User data may not be passed between users during either connection establishment or abortive release.
- The transport provider does not support protocol-specific options. Because these characteristics are not needed by the user, NULL is specified in the third argument to `t_open`. If the user needed a service other than T_COTS_ORD, another transport provider would be opened. An example of the T_CLTS service invocation is presented in the *Introduction to Connectionless-Mode Service* section.

The return value of `t_open` is an identifier for the transport endpoint that will be used by all subsequent Transport Interface function calls. This identifier is actually a file descriptor obtained by opening the transport protocol file [see `open`(2)]. The significance of this fact is highlighted in the *A Read/Write Interface* section.

After the transport endpoint is created, the client calls `t_bind` to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address the user would like to bind to the endpoint, and the third argument is set on return from `t_bind` to specify the address that the provider bound.

The address associated with a server's transport endpoint is important, because that is the address used by all clients to access the server. However, the typical client does not care what its own address is, because no other process will try to access it. That is the case in this example, where the second and third arguments to `t_bind` are set to NULL. A NULL second argument will direct the transport provider to choose an address for the user. A NULL third argument indicates that the user does not care what address was assigned to the endpoint.

If either `t_open` or `t_bind` fail, the program will call `t_error`(3N) to print an appropriate error message to `stderr`. If any Transport Interface routine fails, the global integer `t_errno` will be assigned an appropriate transport error value. A set of such error values has been defined (in `<tiuser.h>`) for the Transport Interface, and `t_error` will print an error message corresponding to the value in `t_errno`. This routine is analogous to `perror`(3), which prints an error message based on the value of `errno`. If the error associated

with a transport function is a system error, `t_errno` will be set to TSYSERR, and `errno` will be set to the appropriate value.

The Server

The server in this example must take similar local management steps before communication can begin. The server must establish a transport endpoint through which it will listen for connect indications. The necessary definitions and local management steps are shown below:

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT  -1
#define SRV_ADDR     1   /* server's well known address */

int conn_fd;             /* connection established here */
extern int t_errno;

main()
{

    int listen_fd;       /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/tivc",
      O_RDWR, NULL)) < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }

    /*
    * By assuming that the address is an integer value,
    * this program may not run over another protocol.
    */

    if ((bind = (struct t_bind *)t_alloc(listen_fd,
      T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }

    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(listen_fd, bind, bind) < 0) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }
```

```
    /*
     * Was the correct address bound?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\0);
        exit(4);
    }
```

As with the client, the first step is to call `t_open` to establish a transport endpoint with the desired transport provider. This endpoint, *listen_fd*, will be used to listen for connect indications. Next, the server must bind its well-known address to the endpoint. This address is used by each client to access the server. The second argument to `t_bind` requests that a particular address be bound to the transport endpoint. This argument points to a `t_bind` structure with the following format:

```
struct t_bind {
    struct netbuf addr;
    unsigned qlen;
}
```

where *addr* describes the address to be bound, and *qlen* indicates the maximum outstanding connect indications that may arrive at this endpoint. All Transport Interface structure and constant definitions are found in `<tiuser.h>`.

The address is specified using a `netbuf` structure that contains the following members:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
}
```

where *buf* points to a buffer containing the data, *len* specifies the bytes of data in the buffer, and *maxlen* indicates the maximum bytes the buffer can hold (and need only be set when data is returned to the user by a Transport Interface routine). For the `t_bind` structure, the data pointed to by *buf* identifies a transport address. It is expected that the structure of addresses will vary among each protocol implementation under the Transport Interface. The `netbuf` structure is intended to support any such structure.

If the value of *qlen* is greater than 0, the transport endpoint may be used to listen for connect indications. In such cases, `t_bind` directs the transport provider to immediately begin queueing connect indications destined for the bound address. Furthermore, the value of *qlen* indicates the maximum outstanding connect indications the server wishes to process. The server must respond to each connect indication, either accepting or rejecting the request for connection. An outstanding connect indication is one to which the server has not yet responded. Often, a

server will fully process a single connect indication and respond to it before receiving the next indication. In this case, a value of 1 is appropriate for *qlen*. However, some servers may wish to retrieve several connect indications before responding to any of them. In such cases, *qlen* indicates the maximum number of such outstanding indications the server will process. An example of a server that manages multiple outstanding connect indications is presented in the *Advanced Topics* section.

`t_alloc`(3N) is called to allocate the `t_bind` structure needed by `t_bind`. `t_alloc` takes three arguments. The first is a file descriptor that references a transport endpoint. This is used to access the characteristics of the transport provider [see `t_open`(3N)]. The second argument identifies the appropriate Transport Interface structure to be allocated. The third argument specifies which, if any, `netbuf` buffers should be allocated for that structure. T_ALL specifies that all `netbuf` buffers associated with the structure should be allocated, and will cause the *addr* buffer to be allocated in this example. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The *maxlen* field of this `netbuf` structure will be set to the size of the newly allocated buffer by `t_alloc`. The use of `t_alloc` will help ensure the compatibility of user programs with future releases of the Transport Interface.

The server in this example will process connect indications one at a time, so *qlen* is set to 1. The address information is then assigned to the newly allocated `t_bind` structure. This `t_bind` structure will be used to pass information to `t_bind` in the second argument, and also will be used to return information to the user in the third argument.

On return, the `t_bind` structure will contain the address that was bound to the transport endpoint. If the provider could not bind the requested address (perhaps because it had been bound to another transport endpoint), it will choose another appropriate address.

NOTE *Each transport provider will manage its address space differently. Some transport providers may allow a single transport address to be bound to several transport endpoints, while others may require a unique address per endpoint. The Transport Interface supports either choice. Based on its address management rules, a provider will determine if it can bind the requested address. If not, it will choose another valid address from its address space and bind it to the transport endpoint.*

The server must check the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients will be unable to reach the server.

If `t_bind` succeeds, the provider will begin queueing connect indications. The next phase of communication, connection establishment, is entered.

Connection Establishment

The connection establishment procedures highlight the distinction between clients and servers. The Transport Interface imposes a different set of procedures in this phase for each type of transport user. The client initiates the connection establishment procedure by requesting a connection to a particular server using `t_connect`(3N). The server is then notified of the client's request by calling `t_listen`(3N). The server may either accept or reject the client's request. It will call `t_accept`(3N) to establish the connection, or call `t_snddis`(3N) to reject the request. The client will be notified of the server's decision when `t_connect` completes.

The Transport Interface supports two facilities during connection establishment that may not be supported by all transport providers. The first is the ability to transfer data between the client and server when establishing the connection. The client may send data to the server when it requests a connection. This data will be passed to the server by `t_listen`. Similarly, the server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by `t_open` determines how much data, if any, two users may transfer during connect establishment.

The second optional service supported by the Transport Interface during connection establishment is the negotiation of protocol options. The client may specify protocol options that it would like the transport provider and/or the remote user. The Transport Interface supports both local and remote option negotiation. As discussed earlier, option negotiation is inherently a protocol-specific function. Use of this facility is discouraged if protocol independent software is a goal (see the *Guidelines for Protocol Independence* section).

The Client

Continuing with the client/server example, the steps needed by the client to establish a connection are shown next:

```
/*
 * By assuming that the address is an integer value,
 * this program may not run over another protocol.
 */
if ((sndcall = (struct t_call *)t_alloc(fd,
  T_CALL, T_ADDR)) == NULL) {
    t_error("t_alloc failed");
    exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;

if (t_connect(fd, sndcall, NULL) < 0) {
    t_error("t_connect failed for fd");
    exit(4);
}
```

The `t_connect` call establishes the connection with the server. The first argument to `t_connect` identifies the transport endpoint through which the connection is established, and the second argument identifies the destination server. This argument is a pointer to a `t_call` structure, which has the following format:

```
struct t_call {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
    int sequence;
}
```

addr identifies the address of the server, *opt* may be used to specify protocol-specific options that the client would like to associate with the connection, and *udata* identifies user data that may be sent with the connect request to the server. The *sequence* field has no meaning for `t_connect`.

`t_alloc` is called above to allocate the `t_call` structure dynamically. Once allocated, the appropriate values are assigned. In this example, no options or user data are associated with the `t_connect` call, but the server's address must be set. The third argument to `t_alloc` is set to T_ADDR to indicate that an appropriate `netbuf` buffer should be allocated for the address. The server's address is then assigned to *buf*, and *len* is set accordingly.

The third argument to `t_connect` can be used to return information about the newly established connection to the user, and may retrieve any user data sent by the server in its response to the connect request. It is set to NULL by the client here to indicate that this information is not needed. The connection will be established on successful return of `t_connect`. If the server rejects the connect request, `t_connect` will fail and set `t_errno` to TLOOK.

Event Handling

The TLOOK error has special significance in the Transport Interface. Some Transport Interface routines may be interrupted by an unexpected asynchronous transport event on the given transport endpoint, and TLOOK notifies the user that an event has occurred. As such, TLOOK does not indicate an error with a Transport Interface routine, but the normal processing of that routine will not be performed because of the pending event. The events defined by the Transport Interface are listed here:

T_LISTEN
: A request for a connection, called a connect indication, has arrived at the transport endpoint.

T_CONNECT
: The confirmation of a previously sent connect request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connect request.

T_DATA
: User data has arrived at the transport endpoint.

T_EXDATA
: Expedited user data has arrived at the transport endpoint. Expedited data will be discussed later in this section.

T_DISCONNECT
: A notification that the connection was aborted or that the server rejected a connect request, called a disconnect indication, has arrived at the transport endpoint.

T_ORDREL
: A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.

T_UDERR
: The notification of an error in a previously sent datagram, called a unitdata error indication, has arrived at the transport endpoint (see the *Introduction to Connectionless-Mode Service* section).

It is possible in some states to receive one of several asynchronous events, as described in the state tables of the *State Transitions* section. The `t_look`(3N) routine enables a user to determine what event has occurred if a TLOOK error is returned. The user can then process that event accordingly. In the example, if a connect request is rejected, the event passed to the client will be a disconnect indication. The client will exit if its request is rejected.

The Server

Returning to the example, when the client calls `t_connect`, a connect indication will be generated on the server's listening transport endpoint. The steps required by the server to process the event are presented below. For each client, the server accepts the connect request and spawns a server process to manage the connection.

```
if ((call = (struct t_call *)t_alloc(listen_fd,
  T_CALL, T_ALL)) == NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(5);
}

while (1) {
    if (t_listen(listen_fd, call) < 0) {
        t_error("t_listen failed for listen_fd");
        exit(6);
    }

    if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
        run_server(listen_fd);
}
```

The server will loop forever, processing each connect indication. First, the server calls `t_listen` to retrieve the next connect indication. When one arrives, the server calls `accept_call` to accept the connect request. `accept_call` accepts the connection on an alternate transport endpoint (as discussed below) and returns the value of that endpoint. *conn_fd* is a global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server may continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, `run_server` will spawn a process to manage the connection.

The server allocates a `t_call` structure to be used by `t_listen`. The third argument to `t_alloc`, `T_ALL`, specifies that all necessary buffers should be allocated for retrieving the caller's address, options, and user data. As mentioned earlier, the transport provider in this example does not support the transfer of user data during connection establishment, and also does not support any protocol options. Therefore, `t_alloc` will not allocate buffers for the user data and options. It must, however, allocate a buffer large enough to store the address of the caller. `t_alloc` determines the buffer size from the *addr* characteristic returned by `t_open`. The *maxlen* field of each `netbuf` structure will be set to the size of the newly allocated buffer by `t_alloc` (*maxlen* is 0 for the user data and options buffers).

Using the `t_call` structure, the server calls `t_listen` to retrieve the next connect indication. If one is currently available, it is returned to the server immediately. Otherwise, `t_listen` will block until a connect indication arrives.

NOTE *The Transport Interface supports an asynchronous mode for such routines that will prevent a process from blocking. This feature is discussed in the Advanced Topics section.*

When a connect indication arrives, the server calls `accept_call` to accept the client's request, as follows:

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }

    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }

    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {   /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
              t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
```

```
        exit(11);
    }
    return(resfd);
}
```

`accept_call` takes two arguments. *listen_fd* identifies the transport endpoint where the connect indication arrived, and *call* is a pointer to a `t_call` structure that contains all information associated with the connect indication. The server will first establish another transport endpoint by opening the clone device node of the transport provider and binding an address. As with the client, a NULL value is passed to `t_bind` to specify that the user does not care what address is bound by the provider. The newly established transport endpoint, *resfd*, is used to accept the client's connect request.

The first two arguments of `t_accept` specify the listening transport endpoint and the endpoint where the connection will be accepted respectively. A connection may be accepted on the listening endpoint. However, this would prevent other clients from accessing the server for the duration of that connection.

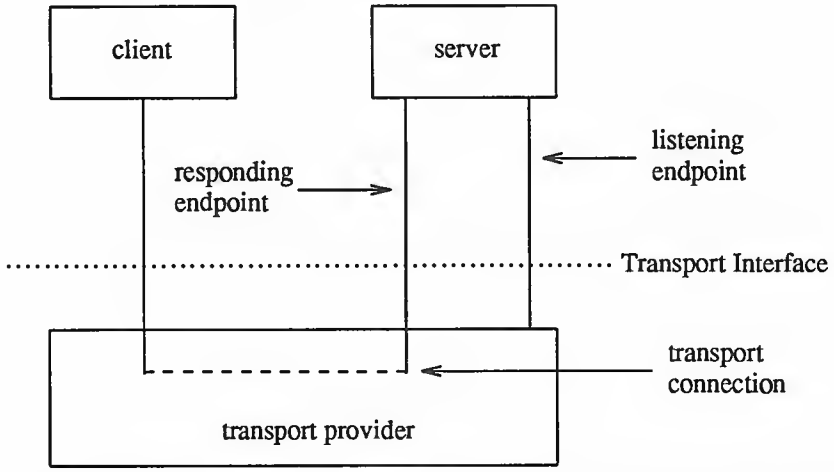
The third argument of `t_accept` points to the `t_call` structure associated with the connect indication. This structure should contain the address of the calling user and the sequence number returned by `t_listen`. The value of *sequence* has particular significance if the server manages multiple outstanding connect indications. The *Advanced Topics* section presents such an example. Also, the `t_call` structure should identify protocol options the user would like to specify, and user data that may be passed to the client. Because the transport provider in this example does not support protocol options or the transfer of user data during connection establishment, the `t_call` structure returned by `t_listen` may be passed without change to `t_accept`.

For simplicity in the example, the server will exit if either the `t_open` or `t_bind` call fails. `exit`(2) will close the transport endpoint associated with *listen_fd*, causing the transport provider to pass a disconnect indication to the client that requested the connection. This disconnect indication notifies the client that the connection was not established; `t_connect` will fail, setting `t_errno` to TLOOK.

`t_accept` may fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted, and `t_errno` will be set to TLOOK. The state transition table in the *State Transitions* section shows that the only event that may occur in this state with only one outstanding connect indication is a disconnect indication. This event may occur if the client decides to undo the connect request it had previously initiated. If a disconnect indication arrives, the server must retrieve the disconnect indication using `t_rcvdis`. This routine takes a pointer to a `t_discon` structure as an argument, which is used to retrieve information associated with a disconnect indication. In this example, however, the server does not care to retrieve this information, so it sets the argument to NULL. After receiving the disconnect indication, `accept_call` closes the responding transport endpoint and returns DISCONNECT, which informs the server that the connection was disconnected by the client. The server then listens for further connect indications.

Figure 9-5 illustrates how the server establishes connections.

Figure 9-5 *Listening and Responding Transport Endpoints*



The transport connection is established on the newly created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

Data Transfer

Once the connection has been established, both the client and server may begin transferring data over the connection using `t_snd` and `t_rcv`. In fact, the Transport Interface does not differentiate the client from the server from this point on. Either user may send and receive data, or release the connection. The Transport Interface guarantees reliable, sequenced delivery of data over an existing connection.

Two classes of data may be transferred over a transport connection: normal and expedited. Expedited data is typically associated with information of an urgent nature. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, all transport protocols do not support the notion of an expedited data class [see `t_open`(3N)].

All transport protocols support the transfer of data in byte stream mode, where "byte stream" implies no concept of message boundaries on data that is transferred over a connection. However, some transport protocols support the preservation of message boundaries over a transport connection. This service is supported by the Transport Interface, but protocol-independent software must not rely on its existence.

The message interface for data transfer is supported by a special flag of `t_snd` and `t_rcv` called T_MORE. The messages, called Transport Service Data Units (TSDU), may be transferred between two transport users as distinct units. The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from `t_open` and `t_getinfo`. Because the maximum TSDU size can be large (possibly unlimited), the Transport Interface enables a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the T_MORE flag on every `t_snd` call except the last. This flag indicates that the user will send more data associated with the message in a subsequent call to `t_snd`. The last message unit should be transmitted with T_MORE turned off to indicate that this is the end of the TSDU.

Similarly, a TSDU may be passed to the user on the receiving side in multiple units. Again, if `t_rcv` returns with the T_MORE flag set, the user should continue calling `t_rcv` to retrieve the remainder of the message. The last unit in the message will be indicated by a call to `t_rcv` that does not set T_MORE.

CAUTION **The T_MORE flag implies nothing about how the data may be packaged below the Transport Interface. Furthermore, it implies nothing about how the data may be delivered to the remote user. Each transport protocol, and each implementation of that protocol, may package and deliver the data differently.**

For example, if a user sends a complete message in a single call to `t_snd`, there is no guarantee that the transport provider will deliver the data in a single unit to the remote transport user. Similarly, a TSDU transmitted in two message units may be delivered in a single unit to the remote transport user. The message boundaries may only be preserved by noting the value of the T_MORE flag on `t_snd` and `t_rcv`. This will guarantee that the receiving user will see a message with the same contents and message boundaries as was sent by the remote user.

The Client

Continuing with the client/server example, the server will transfer a log file to the client over the transport connection. The client receives this data and writes it to its standard output file. A byte stream interface is used by the client and server, where message boundaries (that is, the T_MORE flag) are ignored. The client receives data using the following instructions:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
    if (fwrite(buf, 1, nbytes, stdout) < 0) {
        fprintf(stderr, "fwrite failed\0);
        exit(5);
    }
}
```

The client continuously calls `t_rcv` to process incoming data. If no data is currently available, `t_rcv` blocks until data arrives. `t_rcv` will retrieve the available data up to 1024 bytes, which is the size of the client's input buffer, and will return the number of bytes that were received. The client then writes this data to standard output and continues. The data transfer phase will complete when `t_rcv` fails. `t_rcv` will fail if an orderly release indication or disconnect indication arrives, as will be discussed later in this section. If the `fwrite`(3S) call fails for any reason, the client will exit, thereby closing the transport endpoint. If the transport endpoint is closed (either by `exit` or `t_close`) when it is in the data transfer phase, the connection will be aborted and the remote user will receive a disconnect indication.

The Server

Looking now at the other side of the connection, the server manages its data transfer by spawning a child process to send the data to the client. The parent process then loops back to listen for further connect indications. `run_server` is called by the server to spawn this child process as follows:

```
connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\0);
        exit(12);
    }
    /* else orderly release indication - normal exit */
    exit(0);
}

run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;          /* file pointer to log file */
    char buf[1024];

    switch (fork()) {

    case -1:
        perror("fork failed");
        exit(20);

    default:     /* parent */

        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;

    case 0:      /* child */

        /* close listen_fd and do service */
        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) { /* was disconnect there? */
```

```
            fprintf(stderr, "t_look: unexpected event\0);
            exit(25);
        }

        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
            if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                t_error("t_snd failed");
                exit(26);
            }
```

After the `fork`, the parent process will return to the main processing loop and listen for further connect indications. Meanwhile, the child process will manage the newly established transport connection. If the `fork` call fails, `exit` will close the transport endpoint associated with *listen_fd*. This action will cause a disconnect indication to be passed to the client, and the client's `t_connect` call will fail.

The server process reads 1024 bytes of the log file at a time and sends that data to the client using `t_snd`. *buf* points to the start of the data buffer, and *nbytes* specifies the number of bytes to be transmitted. The fourth argument is used to specify optional flags. Two flags are currently supported: T_EXPEDITED may be set to indicate that the data is expedited, and T_MORE may be set to define message boundaries when transmitting messages over a connection. Neither flag is set by the server in this example.

If the user begins to flood the transport provider with data, the provider may exert back pressure to provide flow control. In such cases, `t_snd` will block until the flow control is relieved, and will then resume its operation. `t_snd` will not complete until *nbyte* bytes have been passed to the transport provider.

The `t_snd` routine does not look for a disconnect indication (signifying that the connection was broken) before passing data to the provider. Also, because the data traffic is flowing in one direction, the user will never look for incoming events. If, for some reason, the connection is aborted, the user should be notified because data may be lost. One option available to the user is to use `t_look` to check for incoming events before each `t_snd` call. A more efficient solution is the one presented in the example. The STREAMS I_SETSIG `ioctl` enables a user to request a signal when a given event occurs [see `streamio`(5) and `signal`(2)]. The STREAMS event of concern here is S_INPUT, which will cause a signal to be sent to the user if any input arrives on the Stream referenced by *conn_fd*. If a disconnect indication arrives, the signal catching routine (`connrelease`) will print an appropriate error message and then exit.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated `t_snd` and `t_rcv` calls, it could rely on `t_rcv` to recognize an incoming disconnect indication.

Connection Release

At any point during data transfer, either user may release the transport connection and end the conversation. As mentioned earlier, two forms of connection release are supported by the Transport Interface. The first, abortive release, breaks a connection immediately and may result in the loss of any data that has not yet reached the destination user. `t_snddis` may be called by either user to generate an abortive release. Also, the transport provider may abort a connection if a problem occurs below the Transport Interface. `t_snddis` enables a user to send data to the remote user when aborting a connection. Although the abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

When the remote user is notified of the aborted connection, `t_rcvdis` must be called to retrieve the disconnect indication. This call will return a reason code that indicates why the connection was aborted, and will return any user data that may have accompanied the disconnect indication (if the abortive release was initiated by the remote user). This reason code is specific to the underlying transport protocol, and should not be interpreted by protocol-independent software.

The second form of connection release is orderly release, which gracefully terminates a connection and guarantees that no data will be lost. All transport providers must support the abortive release procedure, but orderly release is an optional facility that is not supported by all transport protocols.

The Server

The client-server example in this section assumes that the transport provider does support the orderly release of a connection. When all the data has been transferred by the server, the connection may be released as follows:

```
        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();      /* until orderly release indication arrives */
    }
}
```

The orderly release procedure consists of two steps by each user. The first user to complete data transfer may initiate a release using `t_sndrel`, as illustrated in the example. This routine informs the client that no more data will be sent by the server. When the client receives such an indication, it may continue sending data back to the server if desired. When all data has been transferred, however, the client must also call `t_sndrel` to indicate that it is ready to release the connection. The connection will be released only after both users have requested an orderly release and received the corresponding indication from the other user.

In this example, data is transferred in one direction from the server to the client, so the server does not expect to receive data from the client after it has initiated the release procedure. Thus, the server simply calls `pause`(2) after initiating the release. Eventually, the remote user will respond with its orderly release request, and the indication will generate a signal that will be caught by `connrelease`. Remember that the server earlier issued an I_SETSIG `ioctl`

call to generate a signal on any incoming event. Since the only possible Transport Interface events that can occur in this situation are a disconnect indication or orderly release indication, `connrelease` will terminate normally when the orderly release indication arrives. The `exit` call in `connrelease` will close the transport endpoint, thereby freeing the bound address for use by another user. If a user process wants to close a transport endpoint without exiting, it may call `t_close`.

The Client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until `t_rcv` fails. If the server releases the connection (using either `t_snddis` or `t_sndrel`), `t_rcv` will fail and set `t_errno` to TLOOK. The client then processes the connection release as follows:

```
    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}
```

Under normal circumstances, the client terminates the transfer of data by calling `t_sndrel` to initiate the connection release. When the orderly release indication arrives at the client's side of the connection, the client checks to make sure the expected orderly release indication has arrived. If so, it proceeds with the release procedures by calling `t_rcvrel` to process the indication and `t_sndrel` to inform the server that it is also ready to release the connection. At this point the client exits, thereby closing its transport endpoint.

Because all transport providers do not support the orderly release facility just described, users may have to use the abortive release facility provided by `t_snddis` and `t_rcvdis`. However, steps must be taken by each user to prevent any loss of data. For example, a special byte pattern may be inserted in the data stream to indicate the end of a conversation. Many mechanisms are possible for preventing data loss. Each application and high level protocol must choose an appropriate mechanism given the target protocol environment and requirements.

9.5. Introduction to Connectionless-Mode Service

This section describes the connectionless-mode service of the Transport Interface. Connectionless-mode service is appropriate for short-term request/response interactions, such as transaction processing applications. Data are transferred in self-contained units with no logical relationship required among multiple units.

The connectionless-mode services will be described using a transaction server as an example. This server waits for incoming transaction queries, and processes and responds to each query.

Local Management

Just as with connection-mode service, the transport users must perform appropriate local management steps before data can be transferred. A user must choose the appropriate connectionless service provider using `t_open` and establish its identity using `t_bind`.

`t_optmgmt` may be used to negotiate protocol options that may be associated with the transfer of each data unit. As with the connection-mode service, each transport provider specifies the options, if any, that it supports. Option negotiation is therefore a protocol-specific activity.

In the example, the definitions and local management calls needed by the transaction server are as follows:

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR    2        /* server's well known address */

main()
{
    int fd;
    int flags;

    struct t_bind *bind;
    struct t_unitdata *ud;
    struct t_uderr *uderr;

    extern int t_errno;

    if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(fd,
      T_BIND, T_ADDR)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }

    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    bind->qlen = 0;
```

```
    if (t_bind(fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /*
     * is the bound address correct?
     */

    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\0);
        exit(4);
    }
```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using `t_open`. Each provider has an associated service type, so the user may choose a particular service by opening the appropriate transport provider file. This connectionless-mode server ignores the characteristics of the provider returned by `t_open` in the same way as the users in the connection-mode example, setting the third argument to NULL. For simplicity, the transaction server assumes the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the T_CLTS service type (connectionless transport service, or datagram).
- The transport provider does not support any protocol-specific options.

The connectionless server also binds a transport address to the endpoint, so that potential clients may identify and access the server. A `t_bind` structure is allocated using `t_alloc`, and the *buf* and *len* fields of the address are set accordingly.

One important difference between the connection-mode server and this connectionless-mode server is that the *qlen* field of the `t_bind` structure has no meaning for connectionless-mode service. That is because all users are capable of receiving datagrams once they have bound an address. The Transport Interface defines an inherent client-server relationship between two users while establishing a transport connection in the connection-mode service. However, no such relationship exists in the connectionless-mode service. It is the context of this example, not the Transport Interface, that defines one user as a server and another as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by `t_bind` to ensure it is correct.

Data Transfer

Once a user has bound an address to the transport endpoint, datagrams may be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, the Transport Interface enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed earlier, each transport provider defines the set of options, if any, that may accompany a datagram. When the datagram is passed to the destination user, the associated protocol options may be returned as well.

The following sequence of calls illustrates the data transfer phase of the connectionless-mode server:

```
if ((ud = (struct t_unitdata *)t_alloc(fd,
  T_UNITDATA, T_ALL)) == NULL) {
    t_error("t_alloc of t_unitdata structure failed");
    exit(5);
}

if ((uderr = (struct t_uderr *)t_alloc(fd,
  T_UDERROR, T_ALL)) == NULL) {
    t_error("t_alloc of t_uderr structure failed");
    exit(6);
}

while (1) {
    if (t_rcvudata(fd, ud, &flags) < 0) {
        if (t_errno == TLOOK) {

            /*
             * Error on previously sent datagram
             */

            if (t_rcvuderr(fd, uderr) < 0) {
                exit(7);
            }

         fprintf(stderr, "baddatagram, error=%d\n",
                uderr->error);
            continue;
        }
        t_error("t_rcvudata failed");
        exit(8);
    }

    /*
     * Query( ) processes the request and places the
     * response in ud->udata.buf, setting ud->udata.len
     */

    query(ud);

    if (t_sndudata(fd, ud, 0) < 0) {
        t_error("t_sndudata failed");
```

```
                exit(9);
            }
        }
}

query()
{

        /* Merely a stub for simplicity */

}
```

The server must first allocate a `t_unitdata` structure for storing datagrams, which has the following format:

```
struct t_unitdata {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
}
```

addr holds the source address of incoming datagrams and the destination address of outgoing datagrams, *opt* identifies any protocol options associated with the transfer of the datagram, and *udata* holds the data itself. The *addr*, *opt*, and *udata* fields must all be allocated with buffers that are large enough to hold any possible incoming values. As described in the previous section, the T_ALL argument to `t_alloc` will ensure this and will set the *maxlen* field of each `netbuf` structure accordingly. Because the provider does not support protocol options in this example, no options buffer will be allocated, and *maxlen* will be set to zero in the `netbuf` structure for options. A `t_uderr` structure is also allocated by the server for processing any datagram errors, as will be discussed later in this section.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls `t_rcvudata` to receive the next query. `t_rcvudata` will retrieve the next available incoming datagram. If none is currently available, `t_rcvudata` will block, waiting for a datagram to arrive. The second argument of `t_rcvudata` identifies the `t_unitdata` structure where the datagram should be stored.

The third argument, *flags*, must point to an integer variable and may be set to T_MORE on return from `t_rcvudata` to indicate that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to `t_rcvudata` will retrieve the remainder of the datagram. Because `t_alloc` allocates a *udata* buffer large enough to store the maximum datagram size, the transaction server does not have to check the value of *flags*.

If a datagram is received successfully, the transaction server calls the *query* routine to process the request. This routine will store the response in the structure pointed to by *ud*, and will set *ud->udata.len* to indicate the number of bytes in the response. The source address returned by `t_rcvudata` in *ud->addr* will

be used as the destination address by `t_sndudata`.

When the response is ready, `t_sndudata` is called to return the response to the client. The Transport Interface prevents a user from flooding the transport provider with datagrams using the same flow control mechanism described for the connection-mode service. In such cases, `t_sndudata` will block until the flow control is relieved, and will then resume its operation.

Datagram Errors

If the transport provider cannot process a datagram that was passed to it by `t_sndudata`, it will return a unit data error event, T_UDERR, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what may be wrong with the datagram. The reason a datagram could not be processed is protocol-specific. One reason may be that the transport provider could not interpret the destination address or options. Each transport protocol is expected to specify all reasons for which it is unable to process a datagram.

NOTE *The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication will be used. Remember, the connectionless service does not guarantee reliable delivery of data.*

The transaction server will be notified of this error event when it attempts to receive another datagram. In this case, `t_rcvudata` will fail, setting `t_errno` to TLOOK. If TLOOK is set, the only possible event is T_UDERR, so the server calls `t_rcvuderr` to retrieve the event. The second argument to `t_rcvuderr` is the `t_uderr` structure that was allocated earlier. This structure is filled in by `t_rcvuderr` and has the following format:

```
struct t_uderr {
    struct netbuf addr;
    struct netbuf opt;
    long error;
}
```

where *addr* and *opt* identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that indicates why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

9.6. A Read/Write Interface

A user may wish to establish a transport connection and then `exec`(2) an existing user program such as `cat`(1) to process the data as it arrives over the connection. However, existing programs use `read`(2) and `write`(2) for their input/output needs. The Transport Interface does not directly support a `read/write` interface to a transport provider, but one is available with UNIX System V. This interface enables a user to issue `read` and `write` calls over a transport connection that is in the data transfer phase. This section describes the `read/write` interface to the connection-mode service of the Transport Interface. This interface is not available with the connectionless-mode service.

The `read/write` interface is presented using the client example of the *Connection-Mode Client* section with some minor modifications. The clients are identical until the data transfer phase is reached. At that point, this client will use the `read/write` interface and `cat`(1) to process incoming data. `cat` can be run without change over the transport connection. Only the differences between this client and that of the example in the *Connection-Mode Client* section are shown below.

```
#include <stropts.h>
    .
    .    /*
    .     * Same local management and connection
    .     * establishment steps.
    .     */
    .
    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }
    close(0);
    dup(fd);
    execl("/bin/cat", "/bin/cat", 0);
    perror("execl of /bin/cat failed");
    exit(6);
}
```

The client invokes the `read/write` interface by pushing the `tirdwr`(5) module onto the Stream associated with the transport endpoint where the connection was established [see I_PUSH in `streamio`(5)]. This module converts the Transport Interface above the transport provider into a pure `read/write` interface. With the module in place, the client calls `close`(2) and `dup`(2) to establish the transport endpoint as its standard input file, and uses `/bin/cat` to process the input. Because the transport endpoint identifier is a file descriptor, the facility for `dup`ing the endpoint is available to users.

Because the Transport Interface has been implemented using STREAMS, the facilities of this character input/output mechanism can be used to provide enhanced user services. By pushing the `tirdwr` module above the transport provider, the user's interface is effectively changed. The semantics of `read` and `write` must be followed, and message boundaries will not be preserved.

CAUTION The `tirdwr` module may only be pushed onto a Stream when the transport endpoint is in the data transfer phase. Once the module is pushed, the user may not call any Transport Interface routines. If a Transport Interface routine is invoked, `tirdwr` will generate a fatal protocol error, EPROTO, on that Stream, rendering it unusable. Furthermore, if the user pops the `tirdwr` module off the Stream [see I_POP in `streamio`(5)], the transport connection will be aborted.

The exact semantics of `write`, `read`, and `close` using `tirdwr` are described below. To summarize, `tirdwr` enables a user to send and receive

data over a transport connection using `read` and `write`. This module will translate all Transport Interface indications into the appropriate actions. The connection can be released with the `close` system call.

`write`

The user may transmit data over the transport connection using `write`. The `tirdwr` module will pass data through to the transport provider. However, if a user attempts to send a zero-length data packet, which the STREAMS mechanism allows, `tirdwr` will discard the message. If for some reason the transport connection is aborted (for example the remote user aborts the connection using `t_snddis`), a STREAMS hangup condition will be generated on that Stream, and further `write` calls will fail and set *errno* to ENXIO. The user can still retrieve any available data after a hangup, however.

`read`

`read` may be used to retrieve data that has arrived over the transport connection. The `tirdwr` module will pass data through to the user from the transport provider. However, any other event or indication passed to the user from the provider will be processed by `tirdwr` as follows:

- `read` cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, `tirdwr` will generate a fatal protocol error, EPROTO, on that Stream. This error will cause further system calls to fail. You must therefore be aware that you should not communicate with a process that is sending expedited data.
- If an abortive disconnect indication is received, `tirdwr` will discard the indication and generate a STREAMS hangup condition on that Stream. Subsequent `read` calls will retrieve any remaining data, and then `read` will return zero for all further calls (indicating end-of-file).
- If an orderly release indication is received, `tirdwr` will discard the indication and deliver a zero-length STREAMS message to the user. As described in `read`(2), this notifies the user of end-of-file by returning 0 to the user.
- If any other Transport Interface indication is received, `tirdwr` will generate a fatal protocol error, EPROTO, on that Stream. This will cause further system calls to fail. If a user pushes `tirdwr` onto a Stream after the connection has been established, such indications will not be generated.

`close`

With `tirdwr` on a Stream, the user can send and receive data over a transport connection for the duration of that connection. Either user may terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the `tirdwr` module off the Stream. In either case, `tirdwr` will take the following actions:

- If an orderly release indication had previously been received by `tirdwr`, an orderly release request will be passed to the transport provider to complete the orderly release of the connection. The remote user, who initiated the orderly release procedure, will receive the expected indication when data transfer completes.

- If a disconnect indication had previously been received by `tirdwr`, no special action is taken.
- If neither an orderly release indication nor disconnect indication had previously been received by `tirdwr`, a disconnect request will be passed to the transport provider to abortively release the connection.
- If an error had previously occurred on the Stream and a disconnect indication has not been received by `tirdwr`, a disconnect request will be passed to the transport provider.

A process may not initiate an orderly release after `tirdwr` is pushed onto a Stream, but `tirdwr` will handle an orderly release properly if it is initiated by the user on the other side of a transport connection. If the client in this section is communicating with the server program in the *Connection-Mode Client* section, that server will terminate the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. As explained in the first bullet item above, when the file descriptor is closed, `tirdwr` will initiate the orderly release request from the client's side of the connection. This will generate the indication that the server is expecting, and the connection will be released properly.

9.7. Advanced Topics

This section presents important concepts of the Transport Interface that have not been covered in the previous section. First, an optional non-blocking (asynchronous) mode for some library calls is described. Then, an advanced programming example is presented that defines a server that supports multiple outstanding connect indications and operates in an event driven manner.

Asynchronous Execution Mode

Many Transport Interface library routines may block waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, an application may wish to do local processing while waiting for some asynchronous transport interface event.

Support for asynchronous processing of Transport Interface events is available to applications using a combination of the STREAMS asynchronous features (`poll`) and the non-blocking mode of the Transport Interface library routines (I_SETSIG `ioctl`).

In addition, each Transport Interface routine that may block waiting for some event can be run in a special non-blocking mode. For example, `t_listen` will normally block, waiting for a connect indication. However, a server can periodically poll a transport endpoint for existing connect indications by calling `t_listen` in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting O_NDELAY on the file descriptor. This can be set as a flag on `t_open`, or by calling `fcntl`(2) before calling the Transport Interface routine. `fcntl` can be used to enable or disable this mode at any time. All programming examples illustrated throughout this guide use the default, synchronous mode of processing.

O_NDELAY affects each Transport Interface routine in a different manner. To determine the exact semantics of O_NDELAY for a particular routine, see the

appropriate pages in Section 3N of the *SunOS Reference Manual.*

Advanced Programming Example

The following example demonstrates two important concepts. The first is a server's ability to manage multiple outstanding connect indications. The second is an illustration of the ability to write event-driven software using the Transport Interface and the STREAMS system call interface.

The server example in the *Connection-Mode Client* section was capable of supporting only one outstanding connect indication, but the Transport Interface supports the ability to manage multiple outstanding connect indications. One reason a server might wish to receive several, simultaneous connect indications is to impose a priority scheme on each client. A server may retrieve several connect indications, and then accept them in an order based on a priority associated with each client. A second reason for handling several outstanding connect indications is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connect indication, other clients will find it busy. If, however, multiple connect indications can be processed simultaneously, the server will be found to be busy only if the maximum allowed number of clients attempt to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming Transport Interface events, and then takes the appropriate actions for the current event. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and local management functions needed by this example are similar to those of the server example in the *Introduction to Connectionless-Mode Service* section.

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS         1
#define MAX_CONN_IND    4
#define SRV_ADDR        1       /* server's well known address */

int conn_fd;                    /* server connection here */
extern int t_errno;

/* holds connect indications */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;
```

```
/*
 * Only opening and binding one transport endpoint,
 * but more could be supported
 */
if ((pollfds[0].fd = t_open("/dev/tivc",
  O_RDWR, NULL)) < 0) {
    t_error("t_open failed");
    exit(1);
}

if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
  T_BIND, T_ALL)) == NULL) {
    t_error("t_alloc of t_bind structure failed");
    exit(2);
}
bind->qlen = MAX_CONN_IND;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;

if (t_bind(pollfds[0].fd, bind, bind) < 0) {
    t_error("t_bind failed");
    exit(3);
}

/*
 * Was the correct address bound?
 */
if (*(int *)bind->addr.buf != SRV_ADDR) {
    fprintf(stderr, "t_bind bound wrong address0);
    exit(4);
}
```

The file descriptor returned by `t_open` is stored in a `pollfd` structure [see `poll`(2)] that will be used to poll the transport endpoint for incoming data. Notice that only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to the above code.

An important aspect of this server is that it sets *qlen* to a value greater than 1 for `t_bind`. This indicates that the server is willing to handle multiple outstanding connect indications. Remember that the earlier examples single-threaded the connect indications and responses. The server would accept the current connect indication before retrieving additional connect indications. This example, however, can retrieve up to MAX_CONN_IND connect indications at one time before responding to any of them. The transport provider may negotiate the value of *qlen* downward if it cannot support MAX_CONN_IND outstanding connect indications.

Once the server has bound its address and is ready to process incoming connect requests, it does the following:

```
    pollfds[0].events = POLLIN;

    while (1) {
        if (poll(pollfds, NUM_FDS, -1) < 0) {
            perror("poll failed");
            exit(5);
        }

        for (i = 0; i < NUM_FDS; i++) {

            switch (pollfds[i].revents) {

            default:
                perror("poll returned error event");
                exit(6);

            case 0:
                continue;

            case POLLIN:
                do_event(i, pollfds[i].fd);
                service_conn_ind(i, pollfds[i].fd);
            }
        }
    }
}
```

The *events* field of the `pollfd` structure is set to POLLIN, which will notify the server of any incoming Transport Interface events. The server then enters an infinite loop, in which it will `poll` the transport endpoint(s) for events, and then process those events as they occur.

The `poll` call will block indefinitely, waiting for an incoming event. On return, each entry (corresponding to each transport endpoint) is checked for an existing event. If *revents* is set to 0, no event has occurred on that endpoint. In this case, the server continues to the next transport endpoint. If *revents* is set to POLLIN, an event does exist on the endpoint. In this case, `do_event` is called to process the event. If *revents* contains any other value, an error must have occurred on the transport endpoint, and the server will exit.

For each iteration of the loop, if any event is found on the transport endpoint, `service_conn_ind` is called to process any outstanding connect indications. However, if another connect indication is pending, `service_conn_ind` will save the current connect indication and respond to it later. This routine will be explained shortly.

If an incoming event is discovered, the following routine is called to process it:

```
do_event(slot, fd)
{
    struct t_discon *discon;
    int i;

    switch (t_look(fd)) {

    default:
```

```
        fprintf(stderr,"t_look: unexpected event\0);
        exit(7);

    case T_ERROR:
        fprintf(stderr,"t_look returned T_ERROR event\0);
        exit(8);

    case -1:
        t_error("t_look failed");
        exit(9);

    case 0:
        /* since POLLIN returned, this should not happen */
        fprintf(stderr,"t_look returned no event\0);
        exit(10);

    case T_LISTEN:
        /*
         * find free element in calls array
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (calls[slot][i] == NULL)
                break;
        }

        if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
          T_CALL, T_ALL)) == NULL) {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }

        if (t_listen(fd, calls[slot][i]) < 0) {
            t_error("t_listen failed");
            exit(12);
        }
        break;

    case T_DISCONNECT:
        discon = (struct t_discon *)t_alloc(fd,
          T_DIS, T_ALL);

        if (t_rcvdis(fd, discon) < 0) {
            t_error("t_rcvdis failed");
            exit(13);
        }
        /*
         * find call ind in array and delete it
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (discon->sequence ==
              calls[slot][i]->sequence) {
                t_free(calls[slot][i], T_CALL);
                calls[slot][i] = NULL;
            }
        }
        t_free(discon, T_DIS);
        break;
```

```
    }
}
```

This routine takes a number, *slot*, and a file descriptor, *fd*, as arguments. *slot* is used as an index into the global array *calls*. This array contains an entry for each polled transport endpoint, where each entry consists of an array of `t_call` structures that hold incoming connect indications for that transport endpoint. The value of *slot* is used to identify the transport endpoint of interest.

`do_event` calls `t_look` to determine the Transport Interface event that has occurred on the transport endpoint referenced by *fd*. If a connect indication (T_LISTEN event) or disconnect indication (T_DISCONNECT event) has arrived, the event is processed. Otherwise, the server prints an appropriate error message and exits.

For connect indications, `do_event` scans the array of outstanding connect indications looking for the first free entry. A `t_call` structure is then allocated for that entry, and the connect indication is retrieved using `t_listen`. There must always be at least one free entry in the connect indication array, because the array is large enough to hold the maximum number of outstanding connect indications as negotiated by `t_bind`. The processing of the connect indication is deferred until later.

If a disconnect indication arrives, it must correspond to a previously received connect indication. This scenario arises if a client attempts to undo a previous connect request. In this case, `do_event` allocates a `t_discon` structure to retrieve the relevant disconnect information. This structure has the following members:

```
struct t_discon {
    struct netbuf udata;
    int reason;
    int sequence;
}
```

where *udata* identifies any user data that might have been sent with the disconnect indication, *reason* contains a protocol-specific disconnect reason code, and *sequence* identifies the outstanding connect indication that matches this disconnect indication.

Next, `t_rcvdis` is called to retrieve the disconnect indication. The array of connect indications for *slot* is then scanned for one that contains a sequence number that matches the *sequence* number in the disconnect indication. When the connect indication is found, it is freed and the corresponding entry is set to NULL.

As mentioned earlier, if any event is found on a transport endpoint, `service_conn_ind` is called to process all currently outstanding connect indications associated with that endpoint as follows:

```
service_conn_ind(slot, fd)
{
    int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;

        if ((conn_fd = t_open("/dev/tivc", O_RDWR, NULL))
          < 0) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, NULL, NULL) < 0) {
            t_error("t_bind failed");
            exit(15);
        }

        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
}
```

For the given slot (the transport endpoint), the array of outstanding connect indications is scanned. For each indication, the server will open a responding transport endpoint, bind an address to the endpoint, and then accept the connection on that endpoint. If another event (connect indication or disconnect indication) arrives before the current indication is accepted, `t_accept` will fail and set `t_errno` to TLOOK.

NOTE *The user cannot accept an outstanding connect indication if any pending connect indication events or disconnect indication events exist on that transport endpoint.*

If this error occurs, the responding transport endpoint is closed and `service_conn_ind` will return immediately (saving the current connect indication for later processing). This causes the server's main processing loop to be entered, and the new event will be discovered by the next call to `poll`. In this way, multiple connect indications may be queued by the user.

Eventually, all events will be processed, and `service_conn_ind` will be able to accept each connect indication in turn. Once the connection has been established, the `run_server` routine used by the server in the *Connection-Mode Client* section is called to manage the data transfer.

9.8. State Transitions

These tables describe all state transitions associated with the Transport Interface. First, however, the states and events will be described.

Transport Interface States

Table 9-6 defines the states used to describe the Transport Interface state transitions.

Table 9-6 *Transport Interface States*

State	Description	Service Type
T_UNINIT	uninitialized – initial and final state of interface	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	initialized but not bound	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	no connection established	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	outgoing connection pending for client	T_COTS, T_COTS_ORD
T_INCON	incoming connection pending for server	T_COTS, T_COTS_ORD
T_DATAXFER	data transfer	T_COTS, T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Outgoing Events

The outgoing events described in Table 9-7 correspond to the return of the specified transport routines, where these routines send a request or response to the transport provider.

In the figure, some events (such as *acceptN*) are distinguished by the context in which they occur. The context is based on the values of the following variables:

ocnt
count of outstanding connect indications

fd file descriptor of the current transport endpoint

resfd
file descriptor of the transport endpoint where a connection will be accepted

Table 9-7 *Transport Interface Outgoing Events*

Event	Description	Service Type
opened	successful return of `t_open`	T_COTS, T_COTS_ORD, T_CLTS
bind	successful return of `t_bind`	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	successful return of `t_optmgmt`	T_COTS, T_COTS_ORD, T_CLTS
unbind	successful return of `t_unbind`	T_COTS, T_COTS_ORD, T_CLTS
closed	successful return of `t_close`	T_COTS, T_COTS_ORD, T_CLTS
connect1	successful return of `t_connect` in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on `t_connect` in asynchronous mode, or TLOOK error due to a disconnect indication arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	successful return of `t_accept` with *ocnt* == 1, *fd* == *resfd*	T_COTS, T_COTS_ORD
accept2	successful return of `t_accept` with *ocnt* == 1, *fd* != *resfd*	T_COTS, T_COTS_ORD
accept3	successful return of `t_accept` with *ocnt* > 1	T_COTS, T_COTS_ORD
snd	successful return of `t_snd`	T_COTS, T_COTS_ORD
snddis1	successful return of `t_snddis` with *ocnt* <= 1	T_COTS, T_COTS_ORD
snddis2	successful return of `t_snddis` with *ocnt* > 1	T_COTS, T_COTS_ORD
sndrel	successful return of `t_sndrel`	T_COTS_ORD
sndudata	successful return of `t_sndudata`	T_CLTS

Incoming Events

The incoming events correspond to the successful return of the specified routines, where these routines retrieve data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is *pass_conn*, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no Transport Interface routine is issued on that endpoint. *pass_conn* is included in the state tables to describe the behavior when a user accepts a connection on another transport endpoint.

In Table 9-8, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connect indications on the transport endpoint.

Table 9-8 *Transport Interface Incoming Events*

Incoming Event	Description	Service Type
listen	successful return of `t_listen`	T_COTS, T_COTS_ORD
rcvconnect	successful return of `t_rcvconnect`	T_COTS, T_COTS_ORD
rcv	successful return of `t_rcv`	T_COTS, T_COTS_ORD
rcvdis1	successful return of `t_rcvdis` with `ocnt` <= 0	T_COTS, T_COTS_ORD
rcvdis2	successful return of `t_rcvdis` with *ocnt* == 1	T_COTS, T_COTS_ORD
rcvdis3	successful return of `t_rcvdis` with *ocnt* > 1	T_COTS, T_COTS_ORD
rcvrel	successful return of `t_rcvrel`	T_COTS_ORD
rcvudata	successful return of `t_rcvudata`	T_CLTS
rcvuderr	successful return of `t_rcvuderr`	T_CLTS
pass_conn	receive a passed connection	T_COTS, T_COTS_ORD

Transport User Actions

In the state tables that follow, some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [n], where n is the number of the specific action as described below.

[1] Set the count of outstanding connect indications to zero.

[2] Increment the count of outstanding connect indications.

[3] Decrement the count of outstanding connect indications.

[4] Pass a connection to another transport endpoint as indicated in *t_accept*.

State Tables

The following tables describe the Transport Interface state transitions. Given a current state and an event, the transition to the next state is shown, as well as any actions that must be taken by the transport user (indicated by [n]). The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state, given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action list (as specified in the previous section). The transport user must take the specific actions in the order specified in the state table.

The following should be understood when studying the state tables:

- The `t_close` routine is referenced in the state tables (see *closed* event in Table 9-1), but may be called from any state to close a transport endpoint. If `t_close` is called when a transport address is bound to an endpoint, the address will be unbound. Also, if `t_close` is called when the transport connection is still active, the connection will be aborted.

- If a transport user issues a routine out of sequence, the transport provider will recognize this and the routine will fail, setting `t_errno` to TOUTSTATE. The state will not change.
- If any other transport error occurs, the state will not change unless explicitly stated on the manual page for that routine. The exception to this is a TLOOK or TNODATA error on `t_connect`, as described in Table 9-1. The state tables assume correct use of the Transport Interface.
- The support routines `t_getinfo`, `t_getstate`, `t_alloc`, `t_free`, `t_sync`, `t_look`, and `t_error` are excluded from the state tables because they do not affect the state.

A separate table is shown for common local management steps, data transfer in connectionless-mode, and connection-establishment/connection-release/data-transfer in connection-mode.

Figure 9-6 *Common Local Management State Table*

state / event	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Figure 9-7 *Connectionless-Mode State Table*

state / event	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Figure 9-8 *Connection-Mode State Table*

state / event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCON [2]		T_INCON [2]			
accept1			T_DATAXFER[3]			
accept2			T_IDLE [3][4]			
accept3			T_INCON [3][4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE [3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON [3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [3]			
rcvdis3			T_INCON [3]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

9.9. Guidelines for Protocol Independence

By defining a set of services common to many transport protocols, the Transport Interface offers protocol independence for user software. However, all transport protocols do not support all the services supported by the Transport Interface. If software must be run in a variety of protocol environments, only the common services should be accessed. The following guidelines highlight services that may not be common to all transport protocols.

- In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection. If messages must be transferred over a connection, a protocol should be implemented above the Transport Interface to support message boundaries.
- Protocol and implementation specific service limits are returned by the `t_open` and `t_getinfo` routines. These limits are useful when allocating buffers to store protocol-specific transport addresses and options. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.

- User data should not be transmitted with connect requests or disconnect requests [see `t_connect`(3N) and `t_snddis`(3N)]. All transport protocols do not support this capability.
- The buffers in the `t_call` structure used for `t_listen` must be large enough to hold any information passed by the client during connection establishment. The server should use the T_ALL argument to `t_alloc`, which will determine the maximum buffer sizes needed to store the address, options, and user data for the current transport provider.
- The user program should not look at or change options that are associated with any Transport Interface routine. These options are specific to the underlying transport protocol. The user should choose not to pass options with `t_connect` or `t_sndudata`. In such cases, the transport provider will use default values. Also, a server should use the options returned by `t_listen` when accepting a connection.
- Protocol-specific addressing issues should be hidden from the user program. A client should not specify any protocol address on `t_bind`, but instead should allow the transport provider to assign an appropriate address to the transport endpoint. Similarly, a server should retrieve its address for `t_bind` in such a way that it does not require knowledge of the transport provider's address space. Such addresses should not be hard-coded into a program. A name server mechanism could be useful in this scenario, but the details for providing such a service are outside the scope of the Transport Interface.
- The reason codes associated with `t_rcvdis` are protocol-dependent. The user should not interpret this information if protocol-independence is a concern.
- The error codes associated with `t_rcvuderr` are protocol-dependent. The user should not interpret this information if protocol-independence is a concern.
- The names of devices should not be hard-coded into programs, because the device node identifies a particular transport provider, and is not protocol independent.
- The optional orderly release facility of the connection-mode service (provided by `t_sndrel` and `t_rcvrel`) should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. In particular, its use will prevent programs from successfully communicating with ISO open systems.

9.10. Some Examples

The examples presented throughout this guide are shown in entirety in this appendix.

Connection-Mode Client

The following code represents the connection-mode client program described in the *Connection-Mode Client* section. This client establishes a transport connection with a server, and then receives data from the server and writes it to its standard output. The connection is released using the orderly release facility of the Transport Interface. This client will communicate with each of the connection-mode servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR    1    /* server's well known address */

main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }

    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */
    if ((sndcall = (struct t_call *)t_alloc(fd,
      T_CALL, T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }
    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;

    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }

    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
        if (fwrite(buf, 1, nbytes, stdout) < 0) {
            fprintf(stderr, "fwrite failed\0);
```

```
            exit(5);
        }
    }

    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}
```

Connection-Mode Server

The following code represents the connection-mode server program described in the *Connection-Mode Client* section. This server establishes a transport connection with a client, and then transfers a log file to the client on the other side of the connection. The connection is released using the orderly release facility of the Transport Interface. The connection-mode client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT  -1
#define SRV_ADDR     1   /* server's well known address */

int conn_fd;             /* connection established here */
extern int t_errno;

main()
{
    int listen_fd;       /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/tivc", O_RDWR, NULL))
      < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }

    /*
```

```
    * By assuming that the address is an integer value,
    * this program may not run over another protocol.
    */
    if ((bind = (struct t_bind *)t_alloc(listen_fd,
      T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(listen_fd, bind, bind) < 0) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }

    /*
    * Was the correct address bound?
    */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\0);
        exit(4);
    }

    if ((call = (struct t_call *)t_alloc(listen_fd,
      T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(5);
    }

    while (1) {
        if (t_listen(listen_fd, call) < 0) {
            t_error("t_listen failed for listen_fd");
            exit(6);
        }

        if ((conn_fd = accept_call(listen_fd, call))
          != DISCONNECT)
            run_server(listen_fd);
    }
}


accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
```

```
    }

    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }

    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {  /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
              t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}


connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted0);
        exit(12);
    }

    /* else orderly release indication - normal exit */
    exit(0);
}


run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;          /* file pointer to log file */
    char buf[1024];

    switch (fork()) {

    case -1:
        perror("fork failed");
        exit(20);
```

```
    default:     /* parent */

        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;

    case 0:      /* child */

        /* close listen_fd and do service */
        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) { /* was disconnect there? */
            fprintf(stderr, "t_look: unexpected event0);
            exit(25);
        }

        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
            if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                t_error("t_snd failed");
                exit(26);
            }

        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();     /* until orderly release indication arrives */
    }
}
```

Connectionless-Mode Transaction Server

The following code represents the connectionless-mode transaction server program described in the *Introduction to Connectionless-Mode Service* section. This server waits for incoming datagram queries, and then processes each query and sends a response.

```
#include <stdio.h>
```

```
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR    2        /* server's well known address */

main()
{
    int fd;
    int flags;
    struct t_bind *bind;
    struct t_unitdata *ud;
    struct t_uderr *uderr;
    extern int t_errno;


    if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(fd,
      T_BIND, T_ADDR)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    bind->qlen = 0;

    if (t_bind(fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /*
     * is the bound address correct?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\0);
        exit(4);
    }

    if ((ud = (struct t_unitdata *)t_alloc(fd,
      T_UNITDATA, T_ALL)) == NULL) {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
    }
    if ((uderr = (struct t_uderr *)t_alloc(fd,
      T_UDERROR, T_ALL)) == NULL) {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
    }
```

```
    while (1) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
            if (t_errno == TLOOK) {
                /*
                 * Error on previously sent datagram
                 */
                if (t_rcvuderr(fd, uderr) < 0) {
                    t_error("t_rcvuderr failed");
                    exit(7);
                }
                fprintf(stderr, "bad datagram,
                  error = %d\0, uderr->error);
                continue;
            }
            t_error("t_rcvudata failed");
            exit(8);
        }

        /*
         * Query( ) processes the request and places the
         * response in ud->udata.buf, setting ud->udata.len
         */
        query(ud);

        if (t_sndudata(fd, ud, 0) < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}

query()
{
    /* Merely a stub for simplicity */
}
```

Read/Write Client

The following code represents the connection-mode `read/write` client program described in the *A Read/Write Interface* section. This client establishes a transport connection with a server, and then uses `cat`(1) to retrieve the data sent by the server and write it to its standard output. This client will communicate with each of the connection-mode servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>

#define SRV_ADDR    1    /* server's well known address */

main()
{
```

```
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }

    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */

    if ((sndcall = (struct t_call *)t_alloc(fd,
      T_CALL, T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }

    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;

    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }

    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }

    close(0);
    dup(fd);

    execl("/bin/cat", "/bin/cat", 0);

    perror("execl of /bin/cat failed");
    exit(6);
}
```

Event-Driven Server

The following code represents the connection-mode server program described in the *Advanced Topics* section. This server manages multiple connect indications in an event-driven manner. Either connection-mode client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS         1
#define MAX_CONN_IND    4
#define SRV_ADDR        1       /* server's well known address */

int conn_fd;                    /* server connection here */
extern int t_errno;

/* holds connect indications */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;

    /*
     * Only opening and binding one transport endpoint,
     * but more could be supported
     */
    if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR, NULL))
      < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
      T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = MAX_CONN_IND;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(pollfds[0].fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /*
```

```
        * Was the correct address bound?
        */
        if (*(int *)bind->addr.buf != SRV_ADDR) {
            fprintf(stderr, "t_bind bound wrong address\0);
            exit(4);
        }

        pollfds[0].events = POLLIN;

        while (1) {
            if (poll(pollfds, NUM_FDS, -1) < 0) {
                perror("poll failed");
                exit(5);
            }

            for (i = 0; i < NUM_FDS; i++) {

                switch (pollfds[i].revents) {

                default:
                    perror("poll returned error event");
                    exit(6);

                case 0:
                    continue;

                case POLLIN:
                    do_event(i, pollfds[i].fd);
                    service_conn_ind(i, pollfds[i].fd);
                }
            }
        }
}


do_event(slot, fd)
{
        struct t_discon *discon;
        int i;

        switch (t_look(fd)) {

        default:
            fprintf(stderr,"t_look: unexpected event\0);
            exit(7);

        case T_ERROR:
            fprintf(stderr,"t_look returned T_ERROR event\0);
            exit(8);

        case -1:
            t_error("t_look failed");
            exit(9);
```

```
    case 0:
        /* since POLLIN returned, this should not happen */
        fprintf(stderr,"t_look returned no event\0);
        exit(10);

    case T_LISTEN:
        /*
         * find free element in calls array
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (calls[slot][i] == NULL)
                break;
        }

        if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
          T_CALL, T_ALL)) == NULL)
        {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }

        if (t_listen(fd, calls[slot][i]) < 0) {
            t_error("t_listen failed");
            exit(12);
        }
        break;

    case T_DISCONNECT:
        discon = (struct t_discon *)t_alloc(fd,
          T_DIS, T_ALL);

        if (t_rcvdis(fd, discon) < 0) {
            t_error("t_rcvdis failed");
            exit(13);
        }
        /*
         * find call ind in array and delete it
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (discon->sequence ==
              calls[slot][i]->sequence) {
                t_free(calls[slot][i], T_CALL);
                calls[slot][i] = NULL;
            }
        }
        t_free(discon, T_DIS);
        break;
    }
}


service_conn_ind(slot, fd)
{
```

```
    int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;

        if ((conn_fd = t_open("/dev/tivc",
          O_RDWR, NULL)) < 0) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, NULL, NULL) < 0) {
            t_error("t_bind failed");
            exit(15);
        }

        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
}


connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\0);
        exit(12);
    }

    /* else orderly release indication - normal exit */
    exit(0);
}


run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;          /* file pointer to log file */
    char buf[1024];

    switch (fork()) {
```

```
    case -1:
        perror("fork failed");
        exit(20);

    default:    /* parent */

        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;

    case 0:     /* child */

        /* close listen_fd and do service */
        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) { /* disconnect already there? */
            fprintf(stderr, "t_look: unexpected event\0);
            exit(25);
        }

        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
            if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                t_error("t_snd failed");
                exit(26);
            }

        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();    /* until orderly release indication arrives */
    }
}
```

9.11. Glossary

The following terms apply to the Transport Interface:

Abortive release
An abrupt termination of a transport connection, which may result in the loss of data.

Asynchronous execution
The mode of execution in which Transport Interface routines will never block while waiting for specific asynchronous events to occur, but instead will return immediately if the event is not pending.

Client
The transport user in connection-mode that initiates the establishment of a transport connection.

Connection establishment
The phase in connection-mode that enables two transport users to create a transport connection between them.

Connection-mode
A circuit-oriented mode of transfer in which data are passed from one user to another over an established connection in a reliable, sequenced manner.

Connectionless-mode
A mode of transfer in which data are passed from one user to another in self-contained units with no logical relationship required among multiple units.

Connection release
The phase in connection-mode that terminates a previously established transport connection between two users.

Datagram
A unit of data transferred between two users of the connectionless-mode service.

Data transfer
The phase in connection-mode or connectionless-mode that supports the transfer of data between two transport users.

Expedited data
Data that are considered urgent. The specific semantics of *expedited data* are defined by the transport protocol that provides the transport service.

Expedited transport service data
The amount of expedited user data the identity of which is preserved from one end of a transport connection to the other (that is, an expedited message).

Local management
The phase in either connection-mode or connectionless-mode in which a transport user establishes a transport endpoint and binds a transport address to the endpoint. Functions in this phase perform local operations, and require no transport layer traffic over the network.

Orderly release
A procedure for gracefully terminating a transport connection with no loss of data.

Peer user
The user with whom a given user is communicating above the Transport Interface.

Server
The transport user in connection-mode that offers services to other users (clients) and enables these clients to establish a transport connection to it.

Service indication
The notification of a pending event generated by the provider to a user of a particular service.

Service primitive
The unit of information passed across a service interface that contains either a service request or service indication.

Service request
A request for some action generated by a user to the provider of a particular service.

Synchronous execution
The mode of execution in which Transport Interface routines may block while waiting for specific asynchronous events to occur.

Transport address
The identifier used to differentiate and locate specific transport endpoints in a network.

Transport connection
The communication circuit that is established between two transport users in connection-mode.

Transport endpoint
The local communication channel between a transport user and a transport provider.

Transport Interface
The library routines and state transition rules that support the services of a transport protocol.

Transport provider
The transport protocol that provides the services of the Transport Interface.

Transport service data unit
The amount of user data whose identity is preserved from one end of a transport connection to the other (that is, a message).

Transport user
The user-level application or protocol that accesses the services of the Transport Interface.

Virtual circuit
A transport connection established in connection-mode. The following acronyms are used throughout this guide:

CLTS
Connectionless Transport Service

COTS
Connection Oriented Transport Service

ETSDU
Expedited Transport Service Data Unit

TSDU
Transport Service Data Unit

10

A Socket-Based Interprocess Communications Tutorial

WARNING ***Socket-based interprocess communication (IPC), while still supported, is no longer the preferred framework for transport-level programming. Socket-based IPC has been superceded as the "standard" method of accessing network protocols by a set of OSI-compatible transport mechanisms based upon STREAMS and accessed by way of a Transport Library Interface (TLI). For details on the TLI, see the previous chapter, Transport Level Interface Programming.***

If you are building a new network application that requires direct access to transport facilities, use the TLI mechanisms. If you do not require such direct access, Remote Procedure Calls (RPC) are the preferred programming framework — see the ***Remote Procedure Call Programming Guide*** section of this manual for details. New programs should not be based on sockets.

Various approaches are possible within the socket paradigm; this manual discusses them, and then illustrates them by way a series of example programs. These programs demonstrate in a simple way the use of pipes, socketpairs, and the use of datagram socket and stream socket communication.

NOTE *Unlike RPC-based networking (which presumes XDR) socket-based IPC does not contain a mechanism for ensuring architecture independent code. Socket-based programs must make judicious use of the host-to-network byte-order conversion macros described in* `byteorder(3N)` *if they are to be portable.*

The intent of this chapter is to present a few simple example programs, not to describe the socket-based networking facilities in full. For more information, see the next chapter, *An Advanced Socket-Based Interprocess Communications Tutorial.*

10.1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to the UNIX system — first introduced in 4.2BSD. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In the UNIX system a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: creation, use for reading and writing, and destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising

amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the `ls` command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in the UNIX system for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by the UNIX system. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in the UNIX system and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities were implemented in 4.3BSD and then incorporated into SunOS.

Socket-based IPC offers several choices. This chapter presents simple examples that illustrate some of them. The reader is presumed to be familiar with the C programming language, but not necessarily with UNIX system calls or processes and interprocess communication. The chapter reviews the notion of a process and the types of communication that are supported by the socket abstraction. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can serve as models for the programmer trying to construct programs that are composed of cooperating processes.

10.2. Processes

A *process* can be thought of as a single line of control in a program. Programs can have a point where control splits into two independent lines, an action called *forking*. In the UNIX system these lines can never join again. A call to the system routine `fork()` causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value

returned by `fork()`. The parent will receive a process id for the child, the child will receive a zero. Calls to `fork()` typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are the endpoints of communications channels, as discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin*, *stdout*, and *stderr*. These are the standard input, output, and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in the UNIX system), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

10.3. Pipes

Most users of the UNIX system know that they can pipe the output of a program `prog1`, to the input of another, `prog2`, by typing the command

```
example# prog1 | prog2
```

This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example,

```
example# prog1
```

the shell forks a process, which executes the program, `prog1`, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command,

```
example# prog1 | prog2
```

the shell creates two processes connected by a pipe. One process runs the program, `prog1`, the other runs `prog2`, The pipe is an I/O mechanism with two ends. Data that is written into one end can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing `prog1`, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute `prog2` can substitute the opposite end of the pipe for *stdin*.

Now let's examine a program that creates a pipe for communication between its child and itself. A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

Figure 10-1 *Use of a Pipe*

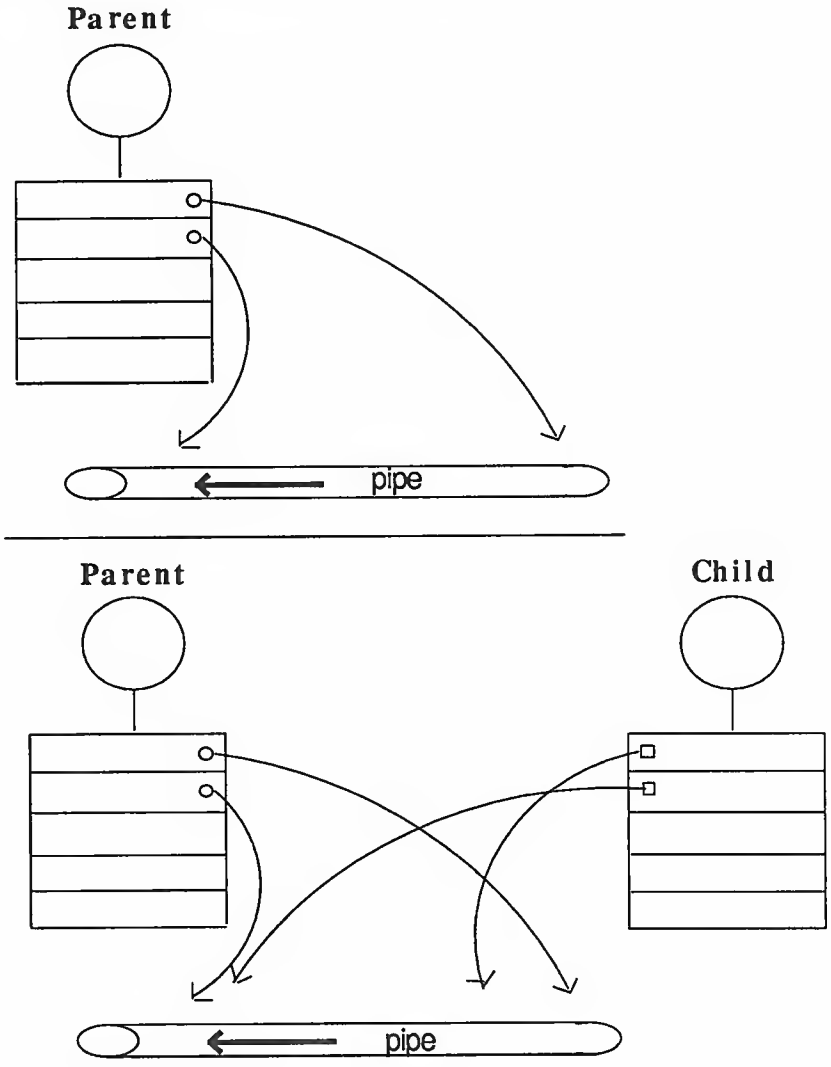
```
#include <stdio.h>
```

```
#define DATA "Bright star, would I . . ."

/*
 * This program creates a pipe, then forks. The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device. I can write to the output socket (sockets[1], the second
 * socket of the array returned by pipe) and read from the input
 * socket (sockets[0]), but not vice versa.
 */

main()
{
    int sockets[2], child;

    /* Create a pipe */
    if (pipe(sockets) < 0) {
        perror("opening stream socket pair");
        exit(10);
    }

    if ((child = fork()) == -1)
        perror("fork");
    else if (child) {
        char buf[1024];

        /* This is still the parent. It reads the child's message. */
        close(sockets[1]);
        if (read(sockets[0], buf, 1024) < 0)
            perror("reading message");
        printf("-->%s\n", buf);
        close(sockets[0]);
    } else {
        /* This is the child. It writes a message to its parent. */
        close(sockets[0]);
        if (write(sockets[1], DATA, sizeof(DATA)) < 0)
            perror("writing message");
        close(sockets[1]);
    }
    exit(0);
}
```

Here the parent process makes a call to the system routine `pipe()`. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. `pipe()`. is passed an array into which it places the index numbers of the sockets it creates. The two ends are not equivalent. The socket whose index is returned in the first word of the array is opened for reading only, while the socket in the second word is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling `fork()`.

The following figure illustrates the effect of such a call to `fork()`. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Figure 10-2 *Sharing a Pipe between Parent and Child*



Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that

is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, it is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to `write()` or from several calls to `write()` that were concatenated.

10.4. Socketpairs

SunOS provides a slight generalization of pipes. A pipe is now a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine `socketpair()`. The program in figure *10-3*, below, calls `socketpair()` to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure *10-4* illustrates the result of a fork following a call to `socketpair()`.

`socketpair()` takes as arguments a specification of a communication domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain specifies a socket name space and implies a set of conventions for manipulating socket names. Currently, socketpairs have only been implemented for the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

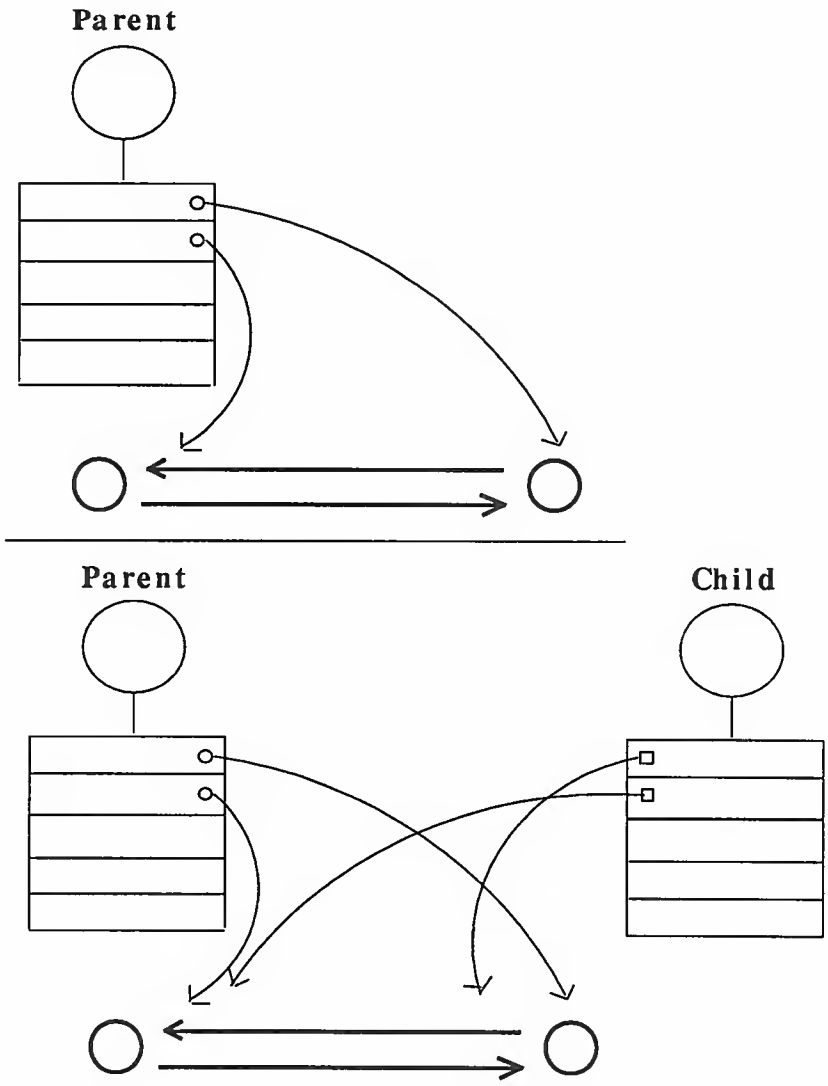
Note that the header files `<sys/socket.h>` and `<sys/types.h>`. are required in this program. The constants `AF_UNIX` and `SOCK_STREAM` are defined in `<sys/socket.h>`, which in turn requires the file `<sys/types.h>` for some of its definitions.

Figure 10-3 *Use of a Socketpair*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects.  Therefore I can
 * send messages in both directions.
 */

main()
{
    int sockets[2], child;
    char buf[1024];

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
```

```
        perror("opening stream socket pair");
        exit(1);
    }

    if ((child = fork()) == -1)
        perror("fork");
    else if (child) {    /* This is the parent */
        close(sockets[0]);
        if (read(sockets[1], buf, 1024, 0) < 0)
            perror("reading stream message");
        printf("-->%s\n", buf);
        if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
            perror("writing stream message");
        close(sockets[1]);
    } else {         /* This is the child */
        close(sockets[1]);
        if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
            perror("writing stream message");
        if (read(sockets[0], buf, 1024, 0) < 0)
            perror("reading stream message");
        printf("-->%s\n", buf);
        close(sockets[0]);
    }
    exit(0);
}
```

Figure 10-4 *Sharing a Socketpair between Parent and Child*



10.5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to communicate between processes that have no common ancestor. Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is specified by a *domain*. There are several such domains for sockets. Two that will be used in the examples here are the UNIX domain (or `AF_UNIX`, for Address Format UNIX) and the Internet domain (or `AF_INET`). In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to it by

giving its pathname. UNIX domain names, thus, allow communication between any two processes that reside on the same machine and that are able to access the socket pathnames. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols `IP/TCP/UDP`. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between separate machines.

Communication follows some particular "style." Currently, communication is either through a *stream* socket or by *datagram*. Stream communication implies a connection. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to `write()` or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

NOTE *Sockets under TLI Emulation:* `write()` *should fail with* `errno` *set to* `ENOTCONN` *if it is used on an unconnected socket, however, under TLI emulation, it will instead return success. Likewise,* `write()` *should fail with* `errno` *set to* `EPIPE` *if a connection is broken, but instead it will return with* `errno` *set to* `ENXIO`. *Similarly,* `read()` *should fail with* `errno` *set to* `ENOTCONN` *if it is used on an unconnected socket, but instead it will return success, with zero bytes read. In all of these cases, however,* `so_error` *will be correctly set. Along the same lines,* `write()`*, should allow zero length data messages on the internet UDP transport. This will not be the case. If it is attempted,* `write()` *will return -1 with* `errno` *set to* `ERANGE`. *These incompatibilities are considered very minor. Note that calling* `send()`, `sendto()` *or* `sendmsg()` *on a CLTS network will succeed.*

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats, and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections, and transfers data between sockets, perhaps sending

the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in figure *10-6* the call to `socket()` causes the creation of a datagram socket with the default protocol in the UNIX domain.

10.6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures *10-5* and *10-6* use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file `<sys/un.h>`. The definition has also been included in the example for clarity.

Each program creates a socket with a call to `socket()`. These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call `bind()`. The program in Figure *10-5* uses the name "socket", which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure *10-6*, use the socket only for sending messages. They do not create a name for the socket because no other process has to refer to it.

Figure 10-5 *Reading UNIX Domain Datagrams*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

/*
 * The include file <sys/un.h> defines sockaddr_un as follows:
 * struct sockaddr_un {
 *   short   sun_family;
 *   char    sun_path[108];
 * };
 */

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
    int sock, length;
    struct sockaddr_un name;
    char buf[1024];

    /* Create socket from which to read. */
```

```
    sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name. */
    name.sun_family = AF_UNIX;
    strcpy(name.sun_path, NAME);
    if (bind(sock, (struct sockaddr *)&name,
      strlen(NAME)+3) < 0) {
        perror("binding name to datagram socket");
        exit(1);
    }
    printf("socket -->%s\n", NAME);
    /* Read from the socket. */
    if (read(sock, buf, 1024) < 0)
        perror("receiving datagram packet");
    printf("-->%s\n", buf);
    close(sock);
    unlink(NAME);
    exit(0);
}
```

Note that, in the call to `bind()` above, the *&name* parameter is cast to a `(struct sockaddr *)`. In writing networking code, one invariably has to cast such address arguments to network-related system calls, since the system-call routines must be able to handle a variety of address formats, yet each individual call will use a specialization of the general format. It is poor programming style to omit these casts, a fact which `lint` will be only to glad to remind you of.

Figure 10-6 *Sending a UNIX Domain Datagrams*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments. The form of the command line is udgramsend pathname.
 */

main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_un name;
```

```
    /* Create socket on which to send. */
    sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Construct name of socket to send to. */
    name.sun_family = AF_UNIX;
    strcpy(name.sun_path, argv[1]);
    /* Send message. */
    if (sendto(sock, DATA, sizeof(DATA), 0,
      (struct sockaddr *)&name,
      sizeof(struct sockaddr_un)) < 0) {
        perror("sending datagram message");
    }
    close(sock);
    exit(0);
}
```

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. "/dev/imaginary") or relative (e.g. "socket"). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (vnode) is allocated in the file system. If the vnode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling `unlink()` or using the `rm(1)` command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure *10-6* gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link.

10.7. Datagrams in the Internet Domain

Figure 10-7 *Reading Internet Domain Datagrams*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * The include file <netinet/in.h> defines sockaddr_in as follows:
 * struct sockaddr_in {
 *    short   sin_family;
 *    u_short sin_port;
 *    struct in_addr sin_addr;
 *    char    sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
    int sock, length;
    struct sockaddr_in name;
    char buf[1024];

    /* Create socket from which to read. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 0;
    if (bind(sock, (struct sockaddr *)&name,
      sizeof name ) < 0) {
        perror("binding datagram socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(name);
    if (getsockname(sock, (struct sockaddr *)&name,
      &length) < 0) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(name.sin_port));
    /* Read from the socket. */
```

```
    if (read(sock, buf, 1024) < 0)
        perror("receiving datagram packet");
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

The examples in Figure *10-7* and *10-8* are very close to the previous examples except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file <netinet/in.h>. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple *<protocol, local machine address, local port, remote machine address, remote port>*. An association may be transient when using datagram sockets; the association actually exists during a `send()` operation.

Figure 10-8 *Sending an Internet Domain Datagram*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is:
 * dgramsend hostname portnumber
 */

main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp, *gethostbyname();
```

```
    /* Create socket on which to send. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to send to.
     * gethostbyname returns a structure including the network address
     * of the specified host. The port number is taken from the command
     * line.
     */
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
        fprintf(stderr, "%s: unknown host0, argv[1]);
        exit(2);
    }
    bcopy((char *)hp->h_addr, (char *)&name.sin_addr,
      hp->h_length);
    name.sin_family = AF_INET;
    name.sin_port = htons(atoi(argv[2]));
    /* Send message. */
    if (sendto(sock, DATA, sizeof DATA , 0,
      (struct sockaddr *)&name, sizeof name) < 0)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value `INADDR_ANY`. The wildcard value is used in the program in Figure *10-7*. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure *10-8* is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to `gethostbyname()`. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit `bind()` call is made with a port number of 0, or when a `connect()` or

`send()` is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling `bind()`, asking for port 0, one may call `getsockname()` to discover what port was actually assigned. The routine `getsockname()` will not work for names in the UNIX domain.

NOTE *Sockets under TLI Emulation:* `getsockname()` *can only work if the underlying transport provider provides the necessary support, and under the TLI, this is not always true. Specifically, if the address given to* `bind()` *was* `INADDR_ANY`*, the the socket module will not be able to map back from its real network address to its local name. This is only a minor problem.*

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by getsockname may result in a misinterpretation. To print out the number, it is necessary to use the routine `ntohs()` (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called `htons()`; similar routines exist for long integers. For further information, see `byteorder(3)`.

10.8. Connections

To send data between stream sockets (having communication style `SOCK_STREAM`), the sockets must be connected. Figures *10-9* and *10-10* show two programs that create such a connection. The program in *10-9* is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls `connect()`, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a `SIGPIPE` signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the `signal(3)` or `sigvec(3)` man pages) the process will terminate.

Figure 10-9 *Initiating an Internet Domain Stream Connection*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line. One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
```

```
 * line is: streamwrite hostname portnumber
 */

main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in server;
    struct hostent *hp, *gethostbyname();
    char buf[1024];

    /* Create socket. */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    server.sin_family = AF_INET;
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
        fprintf(stderr, "%s: unknown host0, argv[1]);
        exit(2);
    }
    bcopy((char *)hp->h_addr, (char *)&server.sin_addr,
      hp->h_length);
    server.sin_port = htons(atoi(argv[2]));

    if (connect(sock,
      (struct sockaddr *)&server, sizeof server ) < 0) {
        perror("connecting stream socket");
        exit(1);
    }
    if (write(sock, DATA, sizeof DATA ) < 0)
        perror("writing on stream socket");
    close(sock);
    exit(0);
}
```

Forming a connection is asymmetrical; one process, such as the program in Figure *10-9* requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure *10-12*. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the

corresponding socket.

The program in Figure *10-10* is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls `listen()` for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. `listen()` marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of `listen()`; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure *10-12* shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The `accept()` call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

NOTE *Sockets under TLI Emulation:* `listen()` *has a unfortunate failure condition under TLI emulation. The problem is rooted in the difference between TLI and socket semantics which creates a timing window within which a second transport user can be allocated the address previously allocated to the caller of* `listen()`. *If this happens, the socket library will return -1, and* `errno` *will be set to* `EADDRINUSE`, *an error not usually possible in sockets. Also note that, both* `read()` *and* `write()` *should return with* `errno` *set to* `ENOCONN` *when used on an unconnected socket. Under the socket emulation, however, they will return success (* `read()` *will also report zero bytes read).* `so_error` *will still be properly set, so these incompatibilities are very minor.*

The program in Figure *10-11* is a slight variation on the server in Figure *10-10*. It avoids blocking when there are no pending connection requests by calling `select()` to check for pending requests before calling `accept()`. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures *10-13* and *10-14* show a program using stream socket communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a machine. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this chapter.

Figure 10-10 *Accepting an Internet Domain Stream Connection*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
    int i;

    /* Create socket. */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Name socket using wildcards. */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = 0;
    if (bind(sock, (struct sockaddr *)&server,
      sizeof server ) < 0) {
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock, (struct sockaddr *)&server,
      &length) < 0) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin_port));

    /* Start accepting connections. */
    listen(sock, 5);
    do {
        msgsock = accept(sock,
            (struct sockaddr *)0, (int *)0);
        if (msgsock == -1)
```

```
            perror("accept");
        else do {
            bzero(buf, sizeof buf );
            if ((rval = read(msgsock, buf, 1024)) < 0)
                perror("reading stream message");
            i = 0;
            if (rval == 0)
                printf("Ending connection\n");
            else
                printf("-->%s\n", buf);
        } while (rval != 0);
        close(msgsock);
    } while (TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock" is
     * never explicitly closed. However, all sockets will be closed
     * automatically when a process is killed or terminates normally.
     */
    exit(0);
}
```

Figure 10-11 *Using* `select()` *to Check for Pending Connections*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select to check that someone is trying to connect
 * before calling accept.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;

    /* Create socket. */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
```

```
    }
    /* Name socket using wildcards. */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = 0;
    if (bind(sock, (struct sockaddr *)&server,
      sizeof server) < 0) {
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock, (struct sockaddr *)&server,
      &length) < 0) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin_port));

    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        if (select(sock + 1, &ready, (fd_set *)0,
          (fd_set *)0, &to) < 0) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0,
              (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                bzero(buf, sizeof buf);
                if ((rval = read(msgsock, buf, 1024)) < 0)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
    } while (TRUE);
    exit(0);
}
```

Figure 10-12 *Establishing a Stream Connection*

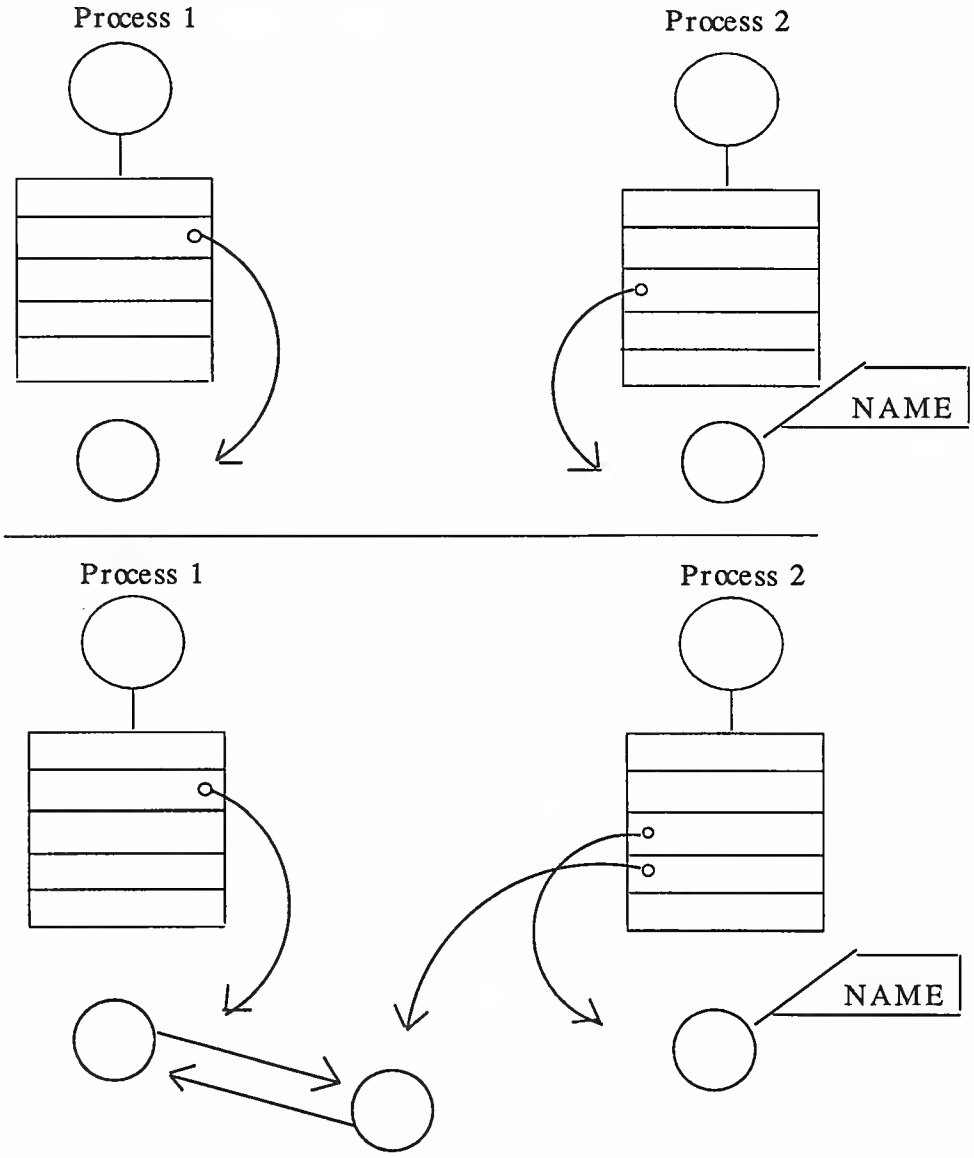


Figure 10-13 *Initiating a UNIX Domain Stream Connection*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is:
 * ustreamwrite pathname
 */
```

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_un server;
    char buf[1024];

    /* Create socket. */
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, argv[1]);

    if (connect(sock, (struct sockaddr *)&server,
      sizeof(struct sockaddr_un)) < 0) {
        close(sock);
        perror("connecting stream socket");
        exit(1);
    }
    if (write(sock, DATA, sizeof(DATA)) < 0)
        perror("writing on stream socket");
    exit(0);
}
```

Figure 10-14 *Accepting a UNIX Domain Stream Connection*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it. When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
    int sock, msgsock, rval;
    struct sockaddr_un server;
    char buf[1024];
```

```
    /* Create socket. */
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Name socket using file system name. */
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, NAME);
    if (bind(sock, (struct sockaddr *)&server,
      sizeof(struct sockaddr_un)) < 0) {
        perror("binding stream socket");
        exit(1);
    }
    printf("Socket has name %s\n", server.sun_path);
    /* Start accepting connections. */
    listen(sock, 5);
    for (;;) {
        msgsock = accept(sock, (struct sockaddr *)0,
          (int *)0);
        if (msgsock == -1)
            perror("accept");
        else do {
            bzero(buf, sizeof buf);
            if ((rval = read(msgsock, buf, 1024)) < 0)
                perror("reading stream message");
            else if (rval == 0)
                printf("Ending connection\n");
            else
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    }
    /*
     * The following statements are not executed, because they follow an
     * infinite loop. However, most ordinary programs will not run
     * forever. In the UNIX domain it is necessary to tell the file
     * system that one is through using NAME. In most programs one uses
     * the call unlink as below. Since the user will have to kill this
     * program, it will be necessary to remove the name by a command from
     * the shell.
     */
    close(sock);
    unlink(NAME);
    exit(0);
}
```

10.9. Reads, Writes, Recvs, etc.

SunOS has several system calls for reading and writing information. The simplest calls are `read()` and `write()`. `write()` takes as arguments the index of a descriptor, a pointer to a buffer containing the data, and the size of the data. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in the *Connections* section below, or a datagram socket for which a `connect`(3) call has provided a default destination. `read()` also takes a descriptor that indicates either a file or a socket. `write()` requires a connected socket since no destination is specified in the parameters of the system call. `read()` can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that make no assumptions about the source of their input or the destination of their output. There are variations on `read()` and `write()` that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are `readv()` and `writev()`, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream socket connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a `SIGURG` signal, if this signal has been enabled (see the `signal(3)` and `sigvec(3)` man pages). See *An Advanced Socket-Based Interprocess Communications Tutorial* for a more complete description of the OOB mechanism.

There are a pair of calls similar to `read()` and `write()` that allow options, including sending and receiving OOB information; these are `send()` and `recv()`. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as `read()` and `write()`.

To send datagrams, one must be allowed to specify the destination. The call `sendto()` takes a destination address as an argument and is therefore used for sending datagrams. The call `recvfrom()` is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use `read()` or `recv()`.

NOTE *Sockets under TLI Emulation: A call to* `recvfrom()` *or* `recvmsg()` *should return the source address if the user supplies a nun-NULL buffer. Under emulation, though, if the user specifies* `MSG_PEEK` *and/or* `MSG_OOB` *then the source address will not be returned. This is only a minor problem.*

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These

are `sendmsg()` and `recvmsg()`. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing, together with their parameters, are shown in Figure *10-15* below. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this chapter, the calls `read()` and `write()` have been used whenever possible.

Figure 10-15 *Varieties of Read and Write Commands*

```
/*
 * The variable descriptor may be the descriptor of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable "sock" must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Note that the meaning assigned to the `msg_accrights` and `msg_accrightslen` fields of the `msghdr` structure used in the `recvmsg()` and `sendmsg()` system calls is protocol-dependent. See the *Scatter/Gather and Exchanging Access Rights* section of the *System Services Overview* for details about the `msghdr` structure.

10.10. Choices

This chapter has presented examples of some of the forms of communication supported by SunOS. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow inter-machine communication.

The two communication domains, the UNIX domain and the Internet domain, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and socket stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that, since a process is only allowed a limited number of open file descriptors (normally 64) there is a limit on the number of streams that a process can have open at any given time. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

10.11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

Further documentation of the socket-based IPC mechanisms can be found in *An Advanced Socket-Based Interprocess Communications Tutorial*. More detailed information about particular calls and protocols is provided in the *SunOS Reference Manual*.

11

An Advanced Socket-Based Interprocess Communications Tutorial

WARNING ***Socket-based interprocess communication (IPC), while still supported, is no longer the preferred framework for transport-level programming. Socket-based IPC has been superceded as the "standard" method of accessing network protocols by a set of OSI-compatible transport mechanisms based upon STREAMS and accessed by way of a Transport Library Interface (TLI). For details on the TLI, see the previous chapter, Transport Level Interface Programming.***

If you are building a new network application that requires direct access to transport facilities, use the TLI mechanisms. If you do not require such direct access, Remote Procedure Calls (RPC) are the preferred programming framework — see the ***Remote Procedure Call Programming Guide*** section of this manual for details. New programs should not be based on sockets.

SunOS contains socket-based IPC mechanisms derived from Berkeley UNIX. This chapter describes the fine points of those mechanisms by supplementing the more introductory information given in *A Socket-Based Interprocess Communications Tutorial*. The majority of the chapter considers the use of these primitives in developing network applications. The reader is expected to be familiar with the C programming language.

Socket-based interprocess communication was first introduced in 4.2BSD and subsequently incorporated into SunOS. The design of these facilities was the result of more than two years of discussion and research, and they incorporated many ideas from then-current research, while maintaining the UNIX philosophy of simplicity and conciseness.

Prior to the 4.2BSD IPC facilities, the only standard mechanism that allowed two processes to communicate were pipes (the mpx files that were in Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of the UNIX system have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the 4.3BSD IPC facilities were designed as a totally independent subsystem. They allow processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a

space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe.

This chapter provides a high-level description of the socket-based IPC facilities and their use. It is designed to complement the manual pages for the IPC primitives with examples of their use. After this initial description, come four more sections. The *Basics* section introduces the IPC-related system calls and the basic model of communication. The *Library Routines* section describes some of the supporting library routines that users may find useful in constructing distributed applications. The *Client/Server Model* section is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. The *Advanced Topics* section delves into advanced topics that sophisticated users are likely to encounter when using the these IPC facilities.

11.1. Basics

The basic building block for communication is the `socket()`. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communications domains*. Domains are abstractions which imply both an addressing structure (address family) and a set of protocols which implement various socket types within the domain (protocol family). Communications domains are introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX domain sockets are named with UNIX path names; e.g. a socket may be named `/dev/foo`. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross between communications domains, but only if some translation process is performed). The 4.3BSD, and thus the socket-based SunOS IPC facilities support several separate communications domains: notably the UNIX domain, for on-system communication, and the Internet domain, which is used by processes that communicate using the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket operating in the UNIX domain sees a subset of the error conditions that are possible when operating in the Internet, DECNET, X.25, or OSI domains.

Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

There are several types of sockets currently available:

- A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes[8].

[8] In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

- A *datagram* socket supports bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which they were sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.
- A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in the *Advanced Topics* section below.

Another potential socket type with interesting properties is the *sequenced packet* socket. Such a socket would have properties similar to those of a stream socket, except that it would preserve record boundaries. There is currently no support for this type of socket.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket.

Socket Creation

To create a socket, the `socket()` system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those that comprise the domain and that may be used to support the requested socket type. The user is returned a descriptor (a small integer number) that may be used in later system calls that operate on sockets. The domain is specified as one of the manifest constants defined in the file `<sys/socket.h>`. For the UNIX domain the constant is

`AF_UNIX`; for the Internet domain, it is `AF_INET`[9]. The socket types are also defined in this file and one of `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW` must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol

[9] The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the `socket()` call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in the *Advanced Topics* section below.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (`ENOBUFS`), a socket request may fail due to a request for an unknown protocol (`EPROTONOSUPPORT`), or a request for a type of socket for which there is no supporting protocol (`EPROTOTYPE`).

Binding Local Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate

<protocol, local address, local port, foreign address, foreign port>

tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate

<protocol, local pathname, foreign pathname>

tuples. Currently, the pathnames may not refer to files already existing on the system, though this may change in future releases.

The `bind()` system call allows a process to specify half of an association,

<local address, local port> (or <local pathname>)

while the `connect()` and `accept()` primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the `connect()` and `send()` calls will automatically bind an appropriate address if they are used with an unbound socket.

The `bind()` system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string that is interpreted by the supporting protocol(s). Its interpretation may vary between communication domains (this is one of the properties that comprise a domain). As mentioned, Internet domain names contain an Internet address and port number. In the UNIX domain, names contain a path name and a family, which is always `AF_UNIX`. If one wanted to bind the name `/tmp/foo` to a UNIX domain socket, the

following code would be used:[10]

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
    sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why `strlen()` is used. In the current implementation of UNIX domain IPC, the file name referred to in `addr.sun_path` is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where `addr.sun_path` is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in the *Library Routines* section when the library routines used in name resolution are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process a *client* and the other a *server*. The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively *listens* on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a *connection* to the server's socket. On the client side the `connect()` call is used to initiate a connection. Using the UNIX domain, this might appear as,

[10] Beware of the tendency to call the "addr" structure "sun", which collides with a symbol predefined by the C preprocessor.

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server,
    strlen(server.sun_path) + sizeof (server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

where *server* in the example above would contain either the UNIX pathname, or the Internet address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. See the *Signals and Process Groups* section below. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (however, any name automatically bound by the system remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the `listen()` call specifies the maximum number of

outstanding connections that may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages that comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may `accept()` a connection:

```
struct sockaddr_in from;
 ...
fromlen = sizeof from;
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a `struct sockaddr_un`, but nothing different would need to be done as far as *fromlen* is concerned. In the examples that follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

`accept()` normally blocks. That is, `accept()` will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in the *Advanced Topics* section below.

Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal `read()` and `write()` system calls are usable,

```
write(s, buf, sizeof buf);
read(s, buf, sizeof buf);
```

In addition to `read()` and `write()`, the calls `send()` and `recv()` may be used:

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

While `send()` and `recv()` are virtually identical to `read()` and `write()`, the extra *flags* argument is important. The flags, defined in `<sys/socket.h>`, may be specified as a non-zero value if one or more of the following is required:

`MSG_OOB`	send/receive out of band data
`MSG_PEEK`	look at data without reading
`MSG_DONTROUTE`	send data without routing packets (internal only)

Out of band data is a notion specific to stream sockets, and one that we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. However, the ability to preview data is of interest. When `MSG_PEEK` is specified with a `recv()` call, any data present is returned to the user, but treated as still "unread". That is, the next `read()` or `recv()` call applied to the socket will return the data previously previewed.

Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a `close()` to the descriptor,

```
close(s);
```

If data is associated with a socket that promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a `shutdown()` on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Connectionless Sockets

To this point we have been concerned mostly with sockets that follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the `bind()` operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the `sendto()` primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message.

When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value `errno` will contain an error number.

To receive messages on an unconnected datagram socket, the `recvfrom()` primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from,
    &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the `from` buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the `connect()` call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (domain `AF_UNSPEC`) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). `accept()` and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send()` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with `getsockopt`, `SO_ERROR`, may be used to interrogate the error status. A `select()` for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in the *Advanced Topics* section below.

Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the `select()` call:

```
#include <sys/time.h>
#include <sys/types.h>
 ...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`select()` takes as arguments pointers to three sets, one for the set of file descriptors on which the caller wishes to be able to read data, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket abstraction. If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the `select()` should be a properly cast null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition `FD_SETSIZE`. The array is long enough to hold one bit for each of `FD_SETSIZE` file descriptors.

The macros `FD_SET` (*fd, &mask*) and `FD_CLR` (*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro `FD_ZERO` (*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the `select()` call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely[11]. `select()` normally returns the number of file descriptors selected; if the `select()` call returns due to the timeout expiring, then the value 0 is returned. If the `select()` terminates because of an error or interruption, a –1 is returned with the error number in `errno`, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the `FD_ISSET` (*fd, &mask*) macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

[11] To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

To determine if there are connections waiting on a socket to be used with an `accept()` call, `select()` can be used, followed by a `FD_ISSET` (*fd, &mask*) macro to check for read readiness on the appropriate socket. If `FD_ISSET` returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
     wait.tv_sec = 1;     /* one second */
     wait.tv_usec = 0;

     FD_ZERO(&read_template);

     FD_SET(s1, &read_template);
     FD_SET(s2, &read_template);

     nb = select(FD_SETSIZE, &read_template, (fd_set *) 0,
          (fd_set *) 0, &wait);
     if (nb <= 0) {
    /*
         * An error occurred during the select, or
     * the select timed out.
     */
     }

     if (FD_ISSET(s1, &read_template)) {
          /* Socket #1 is ready to be read from. */
     }

     if (FD_ISSET(s2, &read_template)) {
          /* Socket #2 is ready to be read from. */
     }
}
```

In previous versions of `select()`, its arguments were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

`select()` provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in the

Advanced Topics section below.

11.2. Library Routines

The discussion in the *Basics* section above indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name that is intended for human consumption; e.g. the *login server* on host monet. This name, and the name of the peer host, must then be translated into network *addresses* that are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file `<netdb.h>` must be included when using any of these routines.

Host Names

An Internet host name to address mapping is represented by the `hostent` structure:

```
struct hostent {
        char *h_name;           /* official name of host */
        char **h_aliases;       /* alias list */
        int  h_addrtype;        /* host address type (e.g., AF_INET) */
        int  h_length;          /* length of address */
        char **h_addr_list;     /* list of addresses, null terminated */
};
#define h_addr h_addr_list[0]   /* first address, network byte order */
```

The routine `gethostbyname(3N)` takes an Internet host name and returns a `hostent` structure, while the routine `gethostbyaddr(3N)` maps Internet host addresses into a *hostent* structure. The routine `inet_ntoa(3N)` maps an Internet host address into an ASCII string for printing by log and error messages.

The official name of the host and its public aliases are returned by these routines, along with the address type (domain) and a null terminated list of variable length addresses. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The `h_addr` definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the *Network Information Service* lookup (the preferred alternative), from the `/etc/hosts` file (see `hosts(5)`), or by use of the `resolver(5)` nameserver. Because of the differences in these databases and their access protocols, the information returned may differ. When using the Network Information Service on the host table version of `gethostbyname()`, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a `netent` structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct netent {
        char *n_name;           /* official name of net */
        char **n_aliases;       /* alias list */
        int  n_addrtype;        /* net address type */
        int  n_net;             /* network number, host byte order */
};
```

The routines `getnetbyname(3N)`, `getnetbynumber(3N)`, and `getnetent(3N)` are the network counterparts to the host routines described above. The routines extract their information from the Network Information

Service maps `hosts.byname` and `hosts.byaddr` or from `/etc/networks`.

Protocol Names

For protocols (which are defined in the Network Information Service `protocols.byname` map and `/etc/protocols`) the `protoent` structure defines the protocol-name mapping used with the routines `getprotobyname`(3N), `getprotobynumber`(3N), and `getprotoent`(3N):

```
struct protoent {
     char *p_name;          /* official protocol name */
     char **p_aliases;      /* alias list */
     int  p_proto;          /* protocol number */
};
```

Service Names

Information regarding services is a bit more complicated. A service is expected to reside at a specific *port* and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the Network Information Service `services.byname` map and the file `/etc/services`. (Actually, the name `services.byname` is a misnomer, since the map actually orders Internet ports by number and protocol).[12] A service mapping is described by the `servent` structure:

```
struct servent {
     char *s_name;          /* official service name */
     char **s_aliases;      /* alias list */
     int  s_port;           /* port number, network byte order */
     char *s_proto;         /* protocol to use */
};
```

The routine `getservbyname(3N)` maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines `getservbyport(3N)` and `getservent(3N)` are also provided. The `getservbyport()` routine has an interface similar to that provided by

[12] For details about the association of RPC services with ports, see the *Port Mapper Program Protocol* section of the *Network Services* chapter.

`getservbyname()`; an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure *11-1*. (This example will be considered in more detail in the *Client/Server Model* section below.)

Aside from the address-related data base routines, there are several other routines available in the run-time library that are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table *11-1* summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Table 11-1 *C Run-time Routines*

Call	*Synopsis*
`bcmp(s1, s2, n)`	Compare byte-strings; 0 if same, not 0 otherwise
`bcopy(s1, s2, n)`	Copy n bytes from s1 to s2
`bzero(base, n)`	Zero-fill n bytes starting at base
`htonl(val)`	32-bit quantity from host into network byte order
`htons(val)`	16-bit quantity from host into network byte order
`ntohl(val)`	32-bit quantity from network into host byte order
`ntohs(val)`	16-bit quantity from network into host byte order

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines that return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines such as the Sun-3 and Sun-4, where these routines are unneeded, they are defined as null macros.[13]

[13] Sun-4 (SPARC) machines do have alignment restrictions which network programmers need to be aware of. See *Porting Software to SPARC Systems*.

Figure 11-1 *Remote Login Client Code*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
    ...
main(argc, argv)
    int argc;
    char *argv[];
{
    struct sockaddr_in server;
    struct servent *sp;
    struct hostent *hp;
    int s;
    ...
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
            "rlogin: tcp/login: unknown service\n");
        exit(1);
    }
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
        fprintf(stderr,
            "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero((char *)&server, sizeof server);
    bcopy(hp->h_addr, (char *)&server.sin_addr,
        hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = sp->s_port;
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
    ...
    /* Connect does the bind for us */

    if (connect(s, (struct sockaddr *)&server,
     sizeof server) < 0) {
        perror("rlogin: connect");
        exit(5);
    }
    ...
    exit(0);
}
```

11.3. Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server that has been examined in the *Basics* section above. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol that must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a *client process* and a *server process*. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes that use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers, this scheme has been implemented via `inetd`, the so called "internet super-server." `inetd` listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which `inetd` is listening, `inetd` executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as `inetd` has played any part in the connection. `inetd` will be described in more detail in the *Advanced Topics* section below.

Servers

In SunOS most servers are accessed at well known Internet addresses or UNIX domain names. The form of their main loop is illustrated by the following code form the remote-login server:

Figure 11-2 *Remote Login Server*

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int f;
    struct sockaddr_in from;
    struct sockaddr_in sin;
    struct servent *sp;
```

```
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
            "rlogind: tcp/login: unknown service\n");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate server from controlling terminal. */
    ...
#endif

    sin.sin_port = sp->s_port;   /* Restricted port */
    sin.sin_addr = INADDR_ANY;
    ...
    f = socket(AF_INET, SOCK_STREAM, 0);
    ...
    if (bind(f, (struct sockaddr *)&sin, sizeof sin) < 0) {
        ...
    }
    ...
    listen(f, 5);
    for (;;) {
        int g, len = sizeof from;

        g = accept(f, (struct sockaddr *) &from, &len);
        if (g < 0) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
        }
        if (fork() == 0) {
            close(f);
            doit(g, &from);
        }
        close(g);
    }
    exit(0);
}
```

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr,
        "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result of the `getservbyname()` call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a

connection). Some standard port numbers are given in the file `/usr/include/netinet/in.h` for backward compatibility purposes.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = getdtablesize()-1; i >= 0; --i)
    close(i);

open("/dev/null", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
    ioctl(i, TIOCNOTTY, 0);
    close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via `syslog()`.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The `bind()` call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a "restricted port number" is covered in the *Advanced Topics* section below.

The main body of the loop is fairly simple:

```
for (;;) {
    int g, len = sizeof from;

    g = accept(f, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    if (fork() == 0) {  /* Child */
        close(f);
        doit(g, &from);
    }
    close(g);          /* Parent */
}
```

An `accept()` call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as `SIGCHLD` (to be discussed in the *Advanced Topics* section below.) Therefore, the return value from `accept()` is checked to insure a connection has actually

been established, and an error report is logged via `syslog()` if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the `accept()` is closed in the parent. The address of the client is also handed the `doit()` routine because it requires it in authenticating clients.

Clients

The client side of the remote login service was shown earlier in Figure *11-1*. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr,
        "rlogin: tcp/login: unknown service\n");
    exit(1);
}
```

Next the destination host is looked up with a `gethostbyname()` call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
    fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
    exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof server);
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that `connect()` implicitly performs a `bind()` call, since `s` is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
    perror("rlogin: socket");
    exit(3);
}
 ...
if (connect(s, (struct sockaddr *)&server,
 sizeof server) < 0) {
    perror("rlogin: connect");
    exit(4);
}
```

The details of the remote login protocol will not be considered here.

Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the `rwho` service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the `ruptime` program. The output generated is illustrated in Figure *11-2*.

Table 11-2 `ruptime` *Output*

arpa	up	9:45,	5 users, load	1.15,	1.39,	1.31
cad	up	2+12:04,	8 users, load	4.67,	5.13,	4.59
calder	up	10:10,	0 users, load	0.27,	0.15,	0.14
dali	up	2+06:28,	9 users, load	1.04,	1.20,	1.65
degas	up	25+09:48,	0 users, load	1.49,	1.43,	1.41
ear	up	5+00:05,	0 users, load	1.51,	1.54,	1.56
ernie	down	0:24				
esvax	down	17:04				
oz	down	16:09				
statvax	up	2+15:57,	3 users, load	1.52,	1.81,	1.86

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The `rwho` server, in a simplified form, is pictured below. It preforms two separate tasks. The first is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error, as a server may be down while a host is actually up.

Figure 11-3 `rwho` *Server*

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
    sin.sin_port = sp->s_port;
    ...
    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on,
     sizeof on) < 0) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    for (;;) {
        struct whod wd;
        int cc, whod, len = sizeof from;

        cc = recvfrom(s, (char *)&wd, sizeof (struct whod),
            0, (struct sockaddr *)&from, &len);
        if (cc <= 0) {
            if (cc < 0 && errno != EINTR)
                syslog(LOG_ERR, "rwhod: recv: %m");
            continue;
        }
        if (from.sin_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                ntohs(from.sin_port));
            continue;
        }
        ...
        if (!verify(wd.wd_hostname)) {
            syslog(LOG_ERR, "rwhod: bad host name from %x",
                ntohl(from.sin_addr.s_addr));
```

```
            continue;
        }
        (void) sprintf(path, "%s/whod.%s", RWHODIR,
            wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *)&wd, cc);
        (void) close(whod);
    }
    exit(0);
}
```

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematic, however.

Status information must be broadcast on the local network. For networks that do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that are not (possibly) directly neighbors. If the server is able to support networks that provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information[14].

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. SunOS attempts to isolate host-specific information from applications by providing system calls that return the necessary information[15]. A mechanism exists, in the form of an `ioctl()` call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been

[14] One must, however, be concerned about *loops*. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

[15] An example of such a system call is the `gethostname(2)` call that returns the host's *official* name.

implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows a solution to the problem of deciding how to propagate status information in the case of `rwho`, or more generally in broadcasting. Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate `ioctl()` calls. The specifics of such broadcastings are complex, however, and will be covered in the *Advanced Topics* section below.

11.4. Advanced Topics

A number of facilities have yet to be discussed. For most programmers, the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features that we consider in this section.

Out Of Band Data

The stream socket abstraction includes the notion of *out of band* data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols (such as TCP) that support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via `MSG_PEEK`) at out of band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the `SIGURG` signal via the appropriate `fcntl()` call, as described below for `SIGIO`. If multiple sockets may have out of band data awaiting delivery, a `select()` call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the `MSG_OOB` flag is supplied to a `send()` or `sendto()` calls, while to receive out of band data `MSG_OOB` should be indicated when performing a `recvfrom()` or `recv()` call. To find out if the read pointer is currently pointing at the mark in the data stream, the `SIOCATMARK` `ioctl` is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is 1 on return, the next read will return data after the mark. Otherwise

(assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in the following example. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

Figure 11-4 *Flushing Terminal I/O on Receipt of Out Of Band Data*

```
#include <sys/ioctl.h>
#include <sys/file.h>
 ...
oob()
{
    int out = FWRITE;
    char waste[BUFSIZ];
    int mark;

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *)&out);
    for (;;) {
        if (ioctl(rem, SIOCATMARK, &mark) < 0) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof waste);
    }
    if (recv(rem, &mark, 1, MSG_OOB) < 0) {
        perror("recv");
        ...
    }
    ...
}
```

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement socket streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a `recv()` is done with the `MSG_OOB` flag. In that case, the call will return an error of `EWOULDBLOCK`. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., `telnet(1C)`) need to retain the position of urgent data within the socket stream. This treatment is available as a socket-level option, `SO_OOBINLINE`; see `setsockopt(2)` for usage. With this option, the

position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Non-Blocking Sockets

It is occasionally convenient to make use of sockets that do not block; that is, I/O requests that cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the `socket()` call, it may be marked as non-blocking by `fcntl()` as follows:

```
#include <fcntl.h>
 ...
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable `errno`), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, `accept()`, `connect()`, `send()`, `recv()`, `read()`, and `write()` can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a `send()` cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

Interrupt Driven Socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the `signal()` or `sigvec()` calls. Second, it must set the process id or process group id that is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This can be accomplished by use of an `fcntl()` call. Third, it must enable asynchronous notification of pending I/O requests with another `fcntl()` call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket `s` is given in Figure *11-5*. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

Figure 11-5 *Use of Asynchronous Notification of I/O Requests*

```
#include <fcntl.h>
 ...
int io_handler();
 ...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us. */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}

/* Allow receipt of asynchronous I/O signals. */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
    perror("fcntl F_SETFL, FASYNC");
    exit(1);
}
```

Signals and Process Groups

Due to the existence of the `SIGURG` and `SIGIO` signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the `F_SETOWN fcntl()`, such as was done in the code above for `SIGIO`. To set the socket's process id for signals, positive arguments should be given to the `fcntl()` call. To set the socket's process group for signals, negative arguments should be passed to `fcntl()`. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar `fcntl()`, `F_GETOWN`, is available for determining the current process number of a socket.

Note that the receipt of `SIGURG` and `SIGIO` can also be enabled by using the `ioctl()` call to assign the socket to the user's process group:

```
. . .
/* oobdata is the out-of-band data handling routine */
signal(SIGURG, oobdata);
. . .
int pid = -getpid();

if (ioctl(client, SIOCSPGRP, (char *)&pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
. . .
```

Another signal that is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop shown in Figure *11-2* may be augmented as shown here:

Figure 11-6 *Use of the SIGCHLD Signal*

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
    int g, len = sizeof from;

    g = accept(f, (struct sockaddr *)&from, &len,);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    ...
}
 ...
#include <wait.h>
reaper()
{
    union wait status;

    while (wait3(&status, WNOHANG, 0) > 0)
        continue;
}
```

If the parent server process fails to reap its children, a large number of *zombie* processes may be created.

Pseudo Terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal.* A pseudo-terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal are supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

The name of the slave side of a pseudo-terminal is of the form `/dev/ttyxy`, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is `/dev/ptyxy`, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal that is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then `fork()`s; the child closes the master side of the pseudo-terminal, and `exec()`s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in the following example. This code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

Figure 11-7 *Creation and Use of a Pseudo Terminal*

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
    line = "/dev/ptyXX";
    line[sizeof "/dev/pty" -1] = c;
    line[sizeof "/dev/ptyp" -1] = '0';
    if (stat(line, &statbuf) < 0)
        break;
    for (i = 0; i < 16; i++) {
        line[sizeof "/dev/ptyp" -1]
            = "0123456789abcdef"[i];
        master = open(line, O_RDWR);
        if (master >= 0) {
            gotpty = 1;
            break;
        }
    }
}
if (!gotpty) {
    syslog(LOG_ERR, "All network ports in use");
    exit(1);
}
```

```
line[sizeof "/dev/" -1] = 't';
slave = open(line, O_RDWR);     /* slave is now slave side */
if (slave < 0) {
    syslog(LOG_ERR, "Cannot open slave pty %s", line);
    exit(1);
}

ioctl(slave, TIOCGETP, &b);     /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
    syslog(LOG_ERR, "fork: %m");
    exit(1);
} else if (i) {         /* Parent */
    close(slave);
    ...
} else {          /* Child */
    close(s);
    close(master);
    dup2(slave, 0);
    dup2(slave, 1);
    dup2(slave, 2);
    if (slave > 2)
        close(slave);
    ...
}
```

Selecting Specific Protocols

If the third argument to the `socket()` call is 0, `socket()` will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet domain may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the protocol domain. For the Internet domain one may use one of the library routines discussed in the *Library Routines* section above, such as `getprotobyname()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with

protocol type of "newtcp" instead of the default "tcp."

Address Binding

As was mentioned in the *Basics* section, binding addresses to sockets in the Internet domain can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the `bind()` system call, a process may specify half of an association, the *<local address, local port>* part, while the `connect()` and `accept()` primitives are used to complete a socket's association by specifying the *<foreign address, foreign port>* part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a *wildcard address* has been provided. When an address is specified as `INADDR_ANY` (a manifest constant defined in `<netinet/in.h>`), the system interprets the address as *any valid address*. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests that are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
    ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system selects the local port number based on two criteria. The first is that Internet ports below `IPPORT_RESERVED` (1024) are reserved for privileged users (i.e., the super user); Internet ports above `IPPORT_USERRESERVED` (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the `rresvport()` library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
    ...
s = rresvport(&lport);
if (s < 0) {
    if (errno == EAGAIN)
        fprintf(stderr, "socket: all ports in use\n");
    else
        perror("rresvport: socket");
    ...
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the `rlogin(1)` command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file `/etc/hosts.equiv` on the system s/he is logging in to (or the system name and the user name are in the user's `.rhosts` file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which s/he is logging in. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the `accept()` call, or from the `getpeername()` call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
 ...
int on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With the above call, local addresses may be bound that are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

Broadcasting and Determining Network Configuration

By using a datagram socket, it is possible to send broadcast packets on many networks connected to the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets that are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in <netinet/in.h>. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, SunOS provides a method of retrieving this information from the system data structures. The SIOCGIFCONF ioctl call returns the interface configuration of a host in the form of a single ifconf structure; this structure contains a "data area" that is made up of an array of ifreq structures, one for each address domain supported by each network interface to which the host is connected. These structures are defined in <net/if.h> as follows:

```
struct ifconf {
    int ifc_len;                    /* size of associated buffer */
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */

struct ifreq {
#define IFNAMSIZ    16
    char ifr_name[IFNAMSIZ];        /* if name, e.g. "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ]; /* other if name */
        short ifru_flags;
        char ifru_data[1];          /* interface dependent data */
    } ifr_ifru;
};

#define ifr_addr    ifr_ifru.ifru_addr    /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr/* other end of link */
#define ifr_oname   ifr_ifru.ifru_oname   /* other if name */
#define ifr_flags   ifr_ifru.ifru_flags   /* flags */
#define ifr_data    ifr_ifru.ifru_data /*for use by interface*/
```

The actual call that obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof buf;
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
    ...
}
```

After this call *buf* will contain a list of `ifreq` structures, one for each network to which the host is connected. These structures will be ordered first by interface name and then by supported address families. `ifc.ifc_len` will have been modified to reflect the number of bytes used by the `ifreq` structures.

For each structure there exists a set of "interface flags" that tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The `SIOCGIFFLAGS ioctl` retrieves these flags for an interface specified by an `ifreq` structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n=ifc.ifc_len/sizeof (struct ifreq);
    --n >= 0; ifr++) {
    /*
     * We must be careful that we don't use an interface
     * devoted to an address domain other than those intended;
     * if we were interested in NS interfaces, the
     * AF_INET would be AF_NS.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /*
     * Skip boring cases
     */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags &
        (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
        continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the `SIOCGIFBRDADDR` `ioctl`, while for point-to-point networks the address of the destination host is obtained with `SIOCGIFDSTADDR`.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
    if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
        sizeof ifr->ifr_dstaddr);
} else if (ifr->ifr_flags & IFF_BROADCAST) {
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
        sizeof ifr->ifr_broadaddr);
}
```

After the appropriate `ioctl()`s have obtained the broadcast or destination address (now in *dst*), the `sendto()` call may be used:

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

In the above loop one `sendto()` occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the sender's address and port, as datagram sockets are bound before a message is allowed to go out.

Socket Options

It is possible to set and get a number of options on sockets via the `setsockopt()` and `getsockopt()` system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level", indicated by the symbolic constant `SOL_SOCKET`, defined in `<sys/socket.h>`. The actual option is specified in `optname`, and is a symbolic constant also defined in `<sys/socket.h>`. *optval* and *optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For `getsockopt()`, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs invoked by `inetd` (described below) may need to perform this task. This can be accomplished as follows via the `SO_TYPE` socket option and the `getsockopt()` call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type,
    &size) < 0) {
    ...
}
```

After the `getsockopt()` call, *type* will be set to the value of the socket type,

as defined in <sys/socket.h>. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

inetd

One of the daemons provided with SunOS is inetd, the so called "Internet super-server." inetd is invoked at boot time and determines from the file /etc/inetd.conf the services for which it is to listen. Once this information has been read and a pristine environment created, inetd proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

inetd then performs a select() on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. inetd then performs an accept() on the socket in question, fork()s, dup()s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and exec()s the appropriate server.

Servers making use of inetd are considerably simplified, as inetd takes care of the majority of the IPC work required in establishing a connection. The server invoked by inetd expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as read(), write(), send(), or recv(). Indeed, servers may use buffered I/O as provided by the "stdio" conventions, as long as they remember to use fflush() when appropriate.

One call that may be of interest to individuals writing servers to be invoked by inetd is the getpeername() call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client connected to a server under inetd, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof name;
 ...
if (getpeername(0,
    (struct sockaddr *)&name, &namelen) < 0) {
    syslog(LOG_ERR, "getpeername: %m");
    exit(1);
} else
    syslog(LOG_INFO, "Connection from %s",
        inet_ntoa(name.sin_addr));
 ...
```

While the getpeername() call is especially useful when writing programs to run with inetd, it can be used under other circumstances. Be warned, however, that getpeername will fail on UNIX domain sockets. ".

12

Socket-Based IPC Implementation Notes

This chapter describes the internal structure of the socket-based networking facilities originally developed for the 4.2BSD version of the UNIX system and subsequently integrated into SunOS. These facilities are based on several central abstractions that structure and unify the external (user) view of network communication as well as the internal (system) implementation. In addition, the implementation introduces a structure for network communications that may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework that promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *System Services Overview*. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, overlap with the interprocess communication tutorials.

Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer, all of the transport and network layers, and some datalink layers.

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data that arrives out of order.

The session layer facilities may provide forms of addressing that are mapped into formats required by the transport layer, service authentication and client

authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface, and other miscellaneous topics.

Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures that could be manipulated by all the pieces of the system, but that required interpretation only by the protocols that "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines that hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

12.1. Memory, Addressing

Address Representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses are internally described by the `sockaddr` structure,

```
struct sockaddr {
    short   sa_family;      /* address family */
    char    sa_data[14];    /* up to 14 bytes of direct address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The `sa_family` field indicates the address family to which the address belongs, and the `sa_data` field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats. Specific address formats use private structure definitions that define the format of the data field. The system interface supports larger address structures, although address-family-independent support facilities, for example routing and raw socket interfaces, provide only 14 bytes for address storage. Protocols that do not use those facilities (e.g, the current UNIX domain) may use larger data areas.[16]

[16] Later versions of the system may support variable length addresses.

Memory Management

A single structure is used for data storage — the memory buffer, or "mbuf". There are three kinds of mbufs — "small", "cluster", and "loaned". They differ in the policies and mechanisms by which their associated storage is allocated and managed.

Small mbufs

Small mbufs are the fundamental type and are used both on their own and as building blocks for cluster and loaned mbufs. They contain their own storage in the array (see below) named `m_dat`. That array is defined as containing 112 (`MLEN`) bytes, so that's all the data that a small mbuf can hold. Small mbufs are guaranteed to start on a 128-byte boundary. The `dtom` macro, described below, works correctly only with small mbufs — mistaken attempts to use `dtom` with cluster and loaned mbufs are a common source of insidious error.

Cluster mbufs

Cluster mbufs support the storage and sharing of larger amounts of data. They do so by dynamically allocating storage, as necessary, from a pool of fixed-sized buffers called clusters. These clusters, each of which is `MCLBYTES` (1K) in size, are managed by the mbuf system itself. The mbuf system uses a small mbuf to refer to a given cluster by setting its `m_off` field to refer to a location in the interior (most commonly, the beginning) of the cluster. This combination of a small mbuf and a cluster is what constitutes a cluster mbuf.

Cluster mbufs can be shared because clusters are reference-counted. The routine `mcldup()` arranges to share an existing cluster mbuf by increasing its reference count and attaching a new small mbuf to it. Cluster mbufs always have their `m_cltype` field set to `MCL_STATIC`.

Loaned mbufs

Loaned mbufs provide for treating storage not directly managed by the mbuf system in the same way as normal mbufs. The mbuf system uses small mbufs to store bookkeeping information about loaned mbufs, as it does with cluster mbufs. With loaned mbufs, however, storage is provided by the allocator, who is ultimately responsible of freeing it as well. To allocate a loaned mbuf, one calls `mclgetx()`, which takes as arguments the address of the buffer to be loaned, its length, a pointer to a function, and an argument to be passed to that function when it's called. This function is called when the loaned mbuf is freed, and must do whatever is necessary to clean up the loaned buffer. The `m_clfun` and `m_clarg` fields of the mbuf structure record the pointer to this function and its argument. Loaned mbufs have

their m_cltype field set to MCL_LOANED.

An mbuf structure has the form:

```
#define MSIZE   128
#define MMINOFF 12
#define MTAIL   4
#define MLEN    (MSIZE-MMINOFF-MTAIL)

struct mbuf {
    struct  mbuf *m_next;          /* next buffer in chain */
    u_long  m_off;                 /* offset of data */
    short   m_len;                 /* amount of data in this mbuf */
    short   m_type;                /* mbuf type (0 == free) */
    union {
            u_char  mun_dat[MLEN];   /* data storage*/
            struct {
                short mun_cltype;    /* "cluster" type*/
                int (*mun_clfun)();
                int mun_clarg;
                int (*mun_clswp)();
            } mun_cl;
    } m_un;
    struct  mbuf *m_act;           /* link in higher-level mbuf list */

#define m_dat   m_un.mun_dat
#define m_cltype    m_un.mun_cl.mun_cltype
#define m_clfun m_un.mun_cl.mun_clfun
#define m_clarg m_un.mun_cl.mun_clarg
};
```

The m_next field is used to chain mbufs together on linked lists, while the m_act field allows lists of mbuf chains to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the m_next field, while groups of objects are linked via the m_act field (possibly when in a queue).

The m_len field indicates the amount of data, while the m_off field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro mtod(), which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define mtod(x,t)    ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, which is used to cast the resultant pointer for proper assignment). Since a small mbuf's data always resides in the mbuf's own m_dat array, its m_off value is always less than MSIZE. On the other hand, storage for cluster and loaned mbufs is external to the mbufs themselves, so their m_off values are always at least MSIZE. The M_HASCL macro distinguishes these two cases and is defined as

```
#define M_HASCL(m)  ((m)->m_off >= MSIZE)
```

As mentioned above, the dtom macro is safe to use only if M_HASCL evaluates false.

The following routines and macros may be used to allocate and free mbufs:

```
m = m_get(wait, type);
MGET(m, wait, type);
```

The subroutine `m_get()` and the macro `MGET()` each allocate an mbuf, placing its address in *m*. The argument *wait* is either `M_WAIT` or `M_DONTWAIT` according to whether allocation should block or fail if no mbuf is available. The *type* is one of the predefined mbuf types for use in accounting of mbuf allocation.

```
MCLGET(m);
```

This macro attempts to allocate an mbuf cluster to associate with the mbuf *m*. If successful, the length of the mbuf is set to `MCLSIZE`. The routine `mclget()` is similar, but returns success/failure.

```
mclgetx(fun, arg, addr, len, wait)
```

This routine wraps the storage defined by *addr* and *len* with an `MCL_LOANED` mbuf. The *fun* argument gives a function to be called when the resulting loaned mbuf is freed, and *arg* is a value that will be supplied to that function as its argument. The argument *wait* is either `M_WAIT` or `M_DONTWAIT` according to whether allocation should block or fail if no mbuf is available.

```
mcldup(m, n, off);
```

A duplicator for cluster and loaned mbufs, which duplicates *m* into *n*. If *m* is a cluster mbuf, `mcldup()` simply bumps its reference count and ignores *off*. But if *m* is a loaned mbuf, `mcldup()` allocates a chunk of memory and copies it, starting at offset *off*.

```
n = m_free(m);
MFREE(m,n);
```

The routine `m_free()` and the macro `MFREE()` each free a single mbuf, *m*, and any associated external storage area, placing a pointer to its successor in the chain it heads, if any, in *n*.

```
m_freem(m);
```

This routine frees an mbuf chain headed by *m*.

By insuring that mbufs always reside on 128 byte boundaries, it is always possible to locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. Note that this works only with objects stored in the internal data buffer of the mbuf. The `dtom` macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define dtom(x) ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets as well as memory allocated for packets and headers. Statistics are maintained on mbuf usage and can be viewed by users using the `netstat()` program. The following utility routines are available for manipulating mbuf chains:

`m = m_copy(m0, off, len);`
The `m_copy()` routine create a copy of all, or part, of a list of the mbufs in *m0*. *len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts are manipulated in preference to core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as *M_COPYALL*, all the data present, offset as before, is copied.

`m_cat(m,n);`
The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

`m_cpytoc(m, off, len, cp)`
Copies a part of the contents of the mbuf *m* to the contiguous memory pointed to by *cp*, skipping the first *off* bytes and copying the next *len* bytes. It returns the number of bytes remaining in the mbuf following the portion copied. *m* is left unaltered.

`m_adj(m, diff);`
The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation; alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

`m = m_pullup(m0, size);`
After a successful call to `m_pullup()`, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory within the data area of the mbuf (allowing access via a pointer, obtained using the `mtod()` macro, and allowing the mbuf to be located from a pointer to the data area using `dtom`, defined below). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single `m_pullup()` call. If the call fails the invalid packet will have been discarded.

12.2. Internal Layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces to which each must conform.

Socket Layer

The socket layer deals with the interprocess communication facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. For more information about the system calls used to manipulate sockets, see *A Socket-Based Interprocess Communications Tutorial* and *An Advanced Socket-Based Interprocess Communications Tutorial*, both sections of *Network Programming*.

A socket consists of the following data structure:

```
struct socket {
    short   so_type;            /* generic type, see socket.h */
    short   so_options;         /* from socket call */
    short   so_linger;          /* time to linger while closing */
    short   so_state;           /* internal state flags SS_*, below */
    caddr_t so_pcb;             /* protocol control block */
    struct  protosw *so_proto;  /* protocol handle */
/*
 * Variables for connection queueing. A socket where accepts occur is so_head
 * in all subsidiary sockets. If so_head is 0, the socket is not related to an
 * accept. For head socket so_q0 queues partially completed connections, while
 * so_q is a queue of connections ready to be accepted. If a connection is
 * aborted and it has so_head set, then it has to be pulled out of either
 * so_q0 or so_q. We allow connections to queue up based on current
 * queue lengths and limit on number of queued connections for this socket.
 */
    struct  socket *so_head;    /* back pointer to accept socket */
    struct  socket *so_q0;      /* queue of partial connections */
    struct  socket *so_q;       /* queue of incoming connections */
    short   so_q0len;           /* partials on so_q0 */
    short   so_qlen;            /* number of connections on so_q */
    short   so_qlimit;          /* max # of queued connections */
    short   so_timeo;           /* connection timeout */
    u_short so_error;           /* error affecting connection */
    short   so_pgrp;            /* pgrp for signals */
    u_short so_oobmark;         /* chars to oob mark */
/*
 * Variables for socket buffering.
 */
    struct  sockbuf  so_rcv;    /* receive buffer  */
    struct  sockbuf  so_snd;    /* send buffer  */
/*
 * Hooks for alternative wakeup strategies.
 * These are used by kernel subsystems wishing to access the socket
 * abstraction. If so_wupfunc is nonnull, it is called in place of
 * wakeup any time that wakeup would otherwise be called with an
 * argument whose value is an address lying within a socket structure.
 */
    struct wupalt   *so_wupalt;
};

struct wupalt {
    int   (*wup_func)();        /* function to call instead of wakeup */
    caddr_t   wup_arg;          /*  argument for so_wupfunc  */
```

```
        /* Other state information here, e.g. for a stream
         * connected to a socket
         */
};
```

Each socket contains two send and receive data queues, `so_rcv` and `so_snd` (see below for a discussion), as well as protocol information, private data, error information and pointers to routines which provide supporting services.

The type of the socket, `so_type` is defined at socket creation time and used in selecting those services that are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the `protosw` structure, which will be described in detail later. A pointer to a protocol-specific data structure, the "protocol control block," is also present in the socket structure. Protocols control this data structure, which normally includes a back pointer to the parent socket structure to allow easy lookup when returning information to a user (for example, placing an error number in the `so_error` field). Other entries in the socket structure are used in queuing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as a "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue."

Socket State

A socket's state is defined from the following:

```
#define SS_NOFDREF              0x001 /* no file table ref any more */
#define SS_ISCONNECTED          0x002 /* socket connected to a peer */
#define SS_ISCONNECTING         0x004 /*in process of connecting to peer*/
#define SS_ISDISCONNECTING      0x008 /* in process of disconnecting */
#define SS_CANTSENDMORE         0x010 /* can't send more data to peer */
#define SS_CANTRCVMORE          0x020 /* can't take more data from peer */
#define SS_RCVATMARK            0x040 /* at mark on input */

#define SS_PRIV                 0x080 /* privileged */
#define SS_NBIO                 0x100 /* non-blocking ops */
#define SS_ASYNC                0x200 /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created, the state is defined based on the type of socket. It may change as control actions are performed, for example connection establishment. It may also change according to the type of input/output the user wishes to perform, as indicated by options set with `fcntl()`. "Non-blocking" I/O implies that a process should never be blocked to await resources. Instead, any call that would block returns prematurely with the error `EWOULDBLOCK`, or

the service request (e.g. a request for more data than is present) may be only partially fulfilled.

If a process requested "asynchronous" notification of events related to the socket, the `SIGIO` signal is posted to the process when such events occur. An event is a change in the socket's state; examples of such occurrences are space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "privileged" if it was created by the super-user. Only privileged sockets may bind addresses in privileged portions of an address space or use "raw" sockets to access lower levels of the network.

Socket Data Queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The structure of a data queue, `struct sockbuf`, is:

```
struct  sockbuf {
    u_short sb_cc;          /* actual chars in buffer */
    u_short sb_hiwat;       /* max actual char count */
    u_short sb_mbcnt;       /* chars of mbufs used */
    u_short sb_mbmax;       /* max chars of mbufs to use */
    u_short sb_lowat;       /* low water mark (not used yet) */
    struct  mbuf *sb_mb;    /* the mbuf chain */
    struct  proc *sb_sel;   /* process selecting read/write */
    short   sb_timeo;       /* timeout (not used yet) */
    short   sb_flags;       /* flags, see below */
} so_rcv, so_snd;
```

Data is stored in a queue as a chain of mbufs. The actual count of data characters as well as high and low water marks are used by the protocols in controlling the flow of data. The amount of buffer space (characters of mbufs and associated data clusters) is also recorded along with the limit on buffer allocation. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).[17]

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources:

```
#define SB_MAX   65535    /* max chars in sockbuf */
#define SB_LOCK  0x01     /* lock on data queue (so_rcv only)*/
#define SB_WANT  0x02     /* someone is waiting to lock */
#define SB_WAIT  0x04     /* someone is waiting for data/space */
#define SB_SEL   0x08     /* buffer is selected */
#define SB_COLL  0x10     /* collision selecting */
```

[17] The low-water mark is always presumed to be 0 in the current implementation.

The last two flags are manipulated by the system in implementing the select mechanism.

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The limit on buffer allocation is set somewhat higher than the limit on data characters to account for the granularity of buffer allocation. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but it is assumed that this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

Data queued at a socket is stored in one of two styles. Stream-oriented sockets queue data with no addresses, headers or record boundaries. The data are in mbufs linked through the `m_next` field. Buffers containing access rights may be present within the chain if the underlying protocol supports passage of access rights. Record-oriented sockets, including datagram sockets, queue data as a list of packets; the sections of packets are distinguished by the types of the mbufs containing them. The mbufs that comprise a record are linked through the `m_next` field; records are linked from the `m_act` field of the first mbuf of one packet to the first mbuf of the next. Each packet begins with an mbuf containing the "from" address if the protocol provides it, then any buffers containing access rights, and finally any buffers containing data. If a record contains no data, no data buffers are required unless neither address nor access rights are present.

Socket Connection Queuing

In dealing with connection oriented sockets (e.g. `SOCK_STREAM`) the two ends are considered distinct. One end is termed *active*, and generates connection requests. The other end is called *passive* and accepts connection requests.

From the passive side, a socket is marked with `SO_ACCEPTCONN` when a `listen()` call is made, creating two queues of sockets: `so_q0` for connections in progress and `so_q` for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on `so_q0` by calling the routine `sonewconn()`. When the connection is established, the socket structure is then transferred to `so_q`, making it available for an `accept()`.

If an `SO_ACCEPTCONN` socket is closed with sockets on either `so_q0` or `so_q`, these sockets are dropped, with notification to the peers as appropriate.

Protocol Layer(s)

Each socket is created in a communications domain, which usually implies both an addressing structure (address family) and a set of protocols that implement various socket types within the domain (protocol family). Each domain is defined by the following structure:

```
struct domain {
        int     dom_family;             /* PF_xxx */
        char    *dom_name;
        int     (*dom_init)();          /* initialize domain structures */
        int     (*dom_externalize)();   /* externalize access rights */
        int     (*dom_dispose)();       /*dispose of internalized rights*/
        struct  protosw *dom_protosw, *dom_protoswNPROTOSW;
        struct  domain *dom_next;
};
```

At boot time, each domain configured into the kernel is added to a linked list of domains. The initialization procedure of each domain is then called. After that time, the domain structure is used to locate protocols within the protocol family. It may also contain procedure references for externalization of access rights at the receiving socket and the disposal of access rights that are not received.

Protocols are described by a set of entry points and certain socket-visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        short   pr_flags;               /* socket visible attributes */
        /* protocol-protocol hooks */
        int     (*pr_input)();          /* input to protocol (from below) */
        int     (*pr_output)();         /* output to protocol (from above) */
        int     (*pr_ctlinput)();       /* control input (from below) */
        int     (*pr_ctloutput)();      /*control output (from above)*/
        /* user-protocol hook */
        int     (*pr_usrreq)();         /* user request */
        /* utility hooks */
        int     (*pr_init)();           /* initialization routine */
        int     (*pr_fasttimo)();       /* fast timeout (200ms) */
        int     (*pr_slowtimo)();       /* slow timeout (500ms) */
        int     (*pr_drain)();          /* flush any excess space possible */
};
```

A protocol is called through the `pr_init` entry before any other. Thereafter it is called every 200 milliseconds through the `pr_fasttimo` entry and every 500 milliseconds through the `pr_slowtimo` for timer based actions. The system will call the `pr_drain` entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the `pr_input()` and `pr_output()` routines. `pr_input()` passes data up

(towards the user) and `pr_output()` passes it down (towards the network); control information passes up and down on `pr_ctlinput()` and `pr_ctloutput()`. The protocol is responsible for the space occupied by any of the arguments to these entries and must either pass it onward or dispose of it. (On output, the lowest level reached must free buffers storing the arguments; on input, the highest level is responsible for freeing buffers.)

The `pr_usrreq()` routine interfaces protocols to the socket code and is described below.

The `pr_flags` field is constructed from the following values:

```
#define PR_ATOMIC           0x01    /* exchange atomic messages only */
#define PR_ADDR             0x02    /* addresses given with messages */
#define PR_CONNREQUIRED     0x04    /* connection required by protocol */
#define PR_WANTRCVD         0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS           0x10    /* passes capabilities */
```

Protocols that are connection-based specify the `PR_CONNREQUIRED` flag so that the socket routines will never attempt to send data before a connection has been established. If the `PR_WANTRCVD` flag is set, the socket routines will notify the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The `PR_ADDR` field indicates that any data placed in the socket's receive queue will be preceded by the address of the sender. The `PR_ATOMIC` flag specifies that each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The `PR_RIGHTS` flag indicates that the protocol supports the passing of capabilities; this is currently used only by the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table for the domain looking for an appropriate protocol to support the type of socket being created. The `pr_type` field contains one of the possible socket types (e.g. `SOCK_STREAM`), while the `pr_domain` is a back pointer to the domain structure. The `pr_protocol` field contains the protocol number of the protocol, normally a well-known value.

Network-Interface Layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and decapsulation of any link-layer header information required to deliver a message to its destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. An interface may have addresses in one or more address families. The address is set at boot time using an `ioctl()` on a socket in the appropriate domain; this operation is implemented by the protocol

family, after verifying the operation through the device `ioctl()` entry.

An interface is defined by the following structure,

```
struct ifnet {
    char    *if_name;              /* name, e.g. "en" or "lo" */
    short    if_unit;              /* sub-unit for lower level driver */
    short    if_mtu;               /* maximum transmission unit */
    short    if_flags;             /* up/down, broadcast, etc. */
    short    if_timer;             /* time 'til if_watchdog called */
    u_short if_promisc;          /* # of requests for promiscuous mode */
    int      if_metric;            /* routing metric (external only) */
    struct  ifaddr *if_addrlist;  /* linked list of addresses per if */
    struct  ifqueue {
        struct  mbuf *ifq_head;
        struct  mbuf *ifq_tail;
        int ifq_len;
        int ifq_maxlen;
        int ifq_drops;
    } if_snd;                      /* output queue */
/* procedure handles */
    int     (*if_init)();          /* init routine */
    int     (*if_output)();        /* output routine */
    int     (*if_ioctl)();         /* ioctl routine */
    int     (*if_reset)();         /* bus reset routine */
    int     (*if_watchdog)();      /* timer routine */
/* generic interface statistics */
    int     if_ipackets;           /* packets received on interface */
    int     if_ierrors;            /* input errors on interface */
    int     if_opackets;           /* packets sent on interface */
    int     if_oerrors;            /* output errors on interface */
    int     if_collisions;         /* collisions on csma interfaces */
/* end statistics */
    struct    ifnet *if_next;
    struct    ifnet *if_upper;     /* next layer up */
    struct    ifnet *if_lower;     /* next layer down */
    int  (*if_input)();            /* input routine */
    int  (*if_ctlin)();            /* control input routine */
    int  (*if_ctlout)();           /* control output routine */
#ifdef sun
    struct map *if_memmap;   /* rmap for interface specific memory */
#endif

};
```

Each interface address has the following form:

```
struct ifaddr {
    struct    sockaddr ifa_addr;     /* address of interface */
    union {
        struct   sockaddr ifu_broadaddr;
        struct   sockaddr ifu_dstaddr;
    } ifa_ifu;
```

```
    struct    ifnet  *ifa_ifp;      /* back-pointer to interface */
    struct    ifaddr *ifa_next;     /* next address for interface */
};
#define ifa_broadaddr ifa_ifu.ifu_broadaddr /*brdcast address*/
#define ifa_dstaddr ifa_ifu.ifu_dstaddr /*other end of link*/
```

The protocol generally maintains this structure as part of a larger structure containing additional information concerning the address.

Each interface has a send queue and routines used for initialization (`if_init`), input (`if_input`), and output (`if_output`). If the interface resides on a system bus, the routine `if_reset` will be called after a bus reset has been performed. An interface may also specify a timer routine, `if_watchdog`; if `if_timer` is non-zero, it is decremented once per second until it reaches zero, at which time the watchdog routine is called.

The state of an interface and certain characteristics are stored in the `if_flags` field. The following values are possible:

```
#define IFF_UP            0x1     /* interface is up */
#define IFF_BROADCAST     0x2     /* broadcast is possible */
#define IFF_DEBUG         0x4     /* turn on debugging */
#define IFF_LOOPBACK      0x8     /* is a loopback net */
#define IFF_POINTOPOINT   0x10    /* interface is point-to-point link */
#define IFF_NOTRAILERS    0x20    /*avoid use of trailers */
#define IFF_RUNNING       0x40i   /* resources allocated */
#define IFF_NOARP         0x80    /* no address resolution protocol  */
#define IFF_PROMISC       0x100   /* receive all packets */
#define IFF_ALLMULTI      0x200   /* receive all multicast packets  */
```

If the interface is connected to a network that supports transmission of *broadcast* packets, the `IFF_BROADCAST` flag will be set and the `ifa_broadaddr` field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point-to-point hardware link (for example, Sunlink/INR), the `IFF_POINTOPOINT` flag will be set and `ifa_dstaddr` will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, `if_addr`, are used in filtering incoming packets. The interface sets `IFF_RUNNING` after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The `IFF_NOTRAILERS` flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets, or (where per-host negotiation of trailers is possible) that trailer encapsulations should not be requested; *trailer* protocols are described in section 14. The `IFF_NOARP` flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses. The `IFF_PROMISC` bit is set when the interface is in *promiscuous mode*, indicating that it should receive all incoming packets regardless of their intended destination.

Various statistics are also stored in the interface structure. These may be viewed by users using the `netstat(1)` program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used initially to define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration. The destination address of a point-to-point link is set with SIOCSIFDSTADDR. Corresponding operations exist to read each value. Protocol families may also support operations to set and read the broadcast address. The SIOCADDMULTI and SCIODELMULTI ioctls may be used to add and remove multicast addresses from the set that the interface accepts. In addition, the SIOCGIFCONF ioctl retrieves a list of interface names and addresses for all interfaces and address families on the host.

12.3. Socket/Protocol Interface

The interface between the socket routines and the communication protocols is through the pr_usrreq() routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH      0   /* attach protocol */
#define PRU_DETACH      1   /* detach protocol */
#define PRU_BIND        2   /* bind socket to address */
#define PRU_LISTEN      3   /* listen for connection */
#define PRU_CONNECT     4   /* establish connection to peer */
#define PRU_ACCEPT      5   /* accept connection from peer */
#define PRU_DISCONNECT  6   /* disconnect from peer */
#define PRU_SHUTDOWN    7   /* won't send any more data */
#define PRU_RCVD        8   /* have taken data; more room now */
#define PRU_SEND        9   /* send this data */
#define PRU_ABORT       10  /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11  /* control operations on protocol */
#define PRU_SENSE       12  /* return status into m */
#define PRU_RCVOOB      13  /* retrieve out of band data */
#define PRU_SENDOOB     14  /* send out of band data */
tdefine PRU_SOCKADDR    15  /* fetch socket's address */
#define PRU_PEERADDR    16  /* fetch peer's address */
#define PRU_CONNECT2    17  /* connect two sockets */
/* begin for protocol's internal use */
#define PRU_FASTTIMO    18  /* 200ms timeout */
#define PRU_SLOWTIMO    19  /* 500ms timeout */
#define PRU_PROTORCV    20  /* receive from below */
#define PRU_PROTOSEND   21  /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(so, req, m, addr, rights);
    int error;
    struct socket *so; int req;
    struct mbuf *m, *addr, *rights;
```

The mbuf data chain *m* is supplied for output operations and for certain other operations where it is to receive a result. The address *addr* is supplied for address-oriented requests such as PRU_BIND and PRU_CONNECT. The *rights* parameter is an optional pointer to an mbuf chain containing user-specified

capabilities (see the `sendmsg()` and `recvmsg()` system calls). The protocol is responsible for disposal of the data mbuf chains on output operations. A non-zero return value gives a UNIX error number that should be passed to higher level software. The following paragraphs describe each of the requests possible.

`PRU_ATTACH`
When a protocol is bound to a socket (with the `socket()` system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

`PRU_DETACH`
This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

`PRU_BIND`
When a socket is initially created it has no address bound to it. This request indicates that an address should be bound to an existing socket. The protocol module must verify that the requested address is valid and available for use.

`PRU_LISTEN`
The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a connection.

`PRU_CONNECT`
The "connect" request indicates the user wants to a establish an association. The `addr` parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel81b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel80], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a `PRU_DISCONNECT` request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

`PRU_ACCEPT`
Following a successful `PRU_LISTEN` request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

`PRU_DISCONNECT`
Eliminate an association created with a `PRU_CONNECT` request.

PRU_SHUTDOWN
: This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the `soshutdown()` system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown and/or notify a connected peer of the shutdown.

PRU_RCVD
: This request is made only if the protocol entry in the protocol switch table includes the `PR_WANTRCVD` flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND
: Each user request to send data is translated into one or more `PRU_SEND` requests (a protocol may indicate that a single user send request must be translated into a single `PRU_SEND` request by specifying the `PR_ATOMIC` flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs, and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT
: This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL
: The "control" request is generated when a user performs a UNIX `ioctl()` system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The `addr` parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual `ioctl()` request code (note the non-standard calling convention). The *rights* parameter contains a pointer to an `ifnet` structure if the `ioctl()` operation pertains to a particular network interface.

PRU_SENSE
: The "sense" request is generated when the user makes an `fstat()` system call on a socket; it requests status of the associated socket. This currently returns a standard `stat()` structure. It typically contains only the optimal transfer size for the connection (based on buffer size, windowing information and maximum packet size). The *m* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed to the protocol module, and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf. An error may be returned if out-of-band data is not

(yet) available or has already been consumed. The *addr* parameter contains any options such as `MSG_PEEK` to examine data without consuming it.

`PRU_SENDOOB`
Like `PRU_SEND`, but for out-of-band data.

`PRU_SOCKADDR`
The local address of the socket is returned, if any is currently bound to it. The address (with protocol specific format) is returned in the *addr* parameter.

`PRU_PEERADDR`
The address of the peer to which the socket is connected is returned. The socket must be in a `SS_ISCONNECTED` state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

`PRU_CONNECT2`
The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the `socketpair(2)` system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the `pr_usrreq` routine solely for convenience in tracing a protocol's operation (e.g. `PRU_SLOWTIMO`).

`PRU_FASTTIMO`
A "fast timeout" has occurred. This request is made when a timeout occurs in the protocol's `pr_fastimo` routine. The *addr* parameter indicates which timer expired.

`PRU_SLOWTIMO`
A "slow timeout" has occurred. This request is made when a timeout occurs in the protocol's `pr_slowtimo()` routine. The *addr* parameter indicates which timer expired.

`PRU_PROTORCV`
This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

`PRU_PROTOSEND`
This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

12.4. Protocol to Protocol Interface

The interface between protocol modules is through the `pr_usrreq()`, `pr_input()`, `pr_output()`, `pr_ctlinput()`, and `pr_ctloutput()` routines. The calling conventions for all but the `pr_usrreq()` routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some

of the Internet protocols in this section as an example.

pr_output()

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
    int error;
    struct inpcb *inp;
    struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on. UDP is based on the Internet Protocol, IP [Postel81a], as its transport. UDP passes a packet to the IP module for output as follows:

```
error = ip_output(m, opt, ro, flags);
    int error;
    struct mbuf *m, *opt;
    struct route *ro; int flags;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is used in making routing decisions (and passing them back to the caller for use in subsequent calls). The final parameter, `flags`, contains flags indicating whether the user is allowed to transmit a broadcast packet and if routing is to be performed. The broadcast flag may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occurred that could be detected immediately (no buffer space available, no route to destination, etc.).

pr_input()

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m, ifp);
    struct mbuf *m;
    struct ifnet *ifp;
```

Each mbuf list passed is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the second parameter.

The IP input routine is a software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, `ipintrq`, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission. The software interrupt

is enabled by the network interfaces when they place input data on the input queue.

pr_ctlinput()

This routine is used to convey "control" information to a protocol module (i.e. information that might be passed to the user, but is not data).

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, addr);
    int req;
    struct sockaddr *addr;
```

The *req* parameter is one of the following,

```
#define   PRC_IFDOWN            0    /* interface transition */
#define   PRC_ROUTEDEAD         1    /* select new route if possible */
#define   PRC_QUENCH            4    /* some said to slow down */
#define   PRC_MSGSIZE           5    /* message size forced drop */
#define   PRC_HOSTDEAD          6    /* normally from IMP */
#define   PRC_HOSTUNREACH       7    /* ditto */
#define   PRC_UNREACH_NET       8    /* no route to network */
#define   PRC_UNREACH_HOST      9    /* no route to host */
#define   PRC_UNREACH_PROTOCOL 10    /* dst says bad protocol */
#define   PRC_UNREACH_PORT     11    /* bad port # */
#define   PRC_UNREACH_NEEDFRAG 12    /* IP_DF caused drop */
#define   PRC_UNREACH_SRCFAIL  13    /* source route failed */
#define   PRC_REDIRECT_NET     14    /* net routing redirect */
#define   PRC_REDIRECT_HOST    15    /* host routing redirect */
#define   PRC_REDIRECT_TOSNET  16    /* redirect for type & net */
#define   PRC_REDIRECT_TOSHOST 17    /* redirect for tos & host */
#define   PRC_TIMXCEED_INTRANS 18    /* packet expired in transit */
#define   PRC_TIMXCEED_REASS   19    /* lifetime expired on reass q */
#define   PRC_PARAMPROB        20    /* header incorrect */
```

while the *addr* parameter is the address to which the condition applies. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol [Postel81c]), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes that are delivered to a user.

pr_ctloutput()

This is the routine that implements per-socket options at the protocol level for `getsockopt()` and `setsockopt()`. The calling convention is,

```
error = (*protosw[].pr_ctloutput)(op,so,level,optname,mp);
    int op;
    struct socket *so;
    int level, optname;
    struct mbuf **mp;
```

where *op* is one of `PRCO_SETOPT` or `PRCO_GETOPT`, *so* is the socket whence

the call originated, and `level` and *optname* are the protocol level and option name supplied by the user. The results of a `PRCO_GETOPT` call are returned in an mbuf whose address is placed in *mp* before return. On a `PRCO_SETOPT` call, *mp* contains the address of an mbuf containing the option data; the mbuf should be freed before return.

12.5. Protocol/Network-Interface Interface

The lowest layer in the set of protocols that comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface; in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases with which to be concerned, transmission of a packet and receipt of a packet; each will be considered separately.

Packet Transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
    int error;
    struct ifnet *ifp;
    struct mbuf *m;
    struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable media, such as the Ethernet, "successful" transmission simply means that the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are only those that can be detected immediately, and are normally trivial in nature (no buffer space, address format not handled, etc.). No indication is received if errors are detected after the call has returned.

Packet Reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In this system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued for the protocol module, and a software interrupt is posted to initiate processing.

Three macros are available for queuing and dequeuing packets:

`IF_ENQUEUE(ifq, m)`
This places the packet *m* at the tail of the queue *ifq*.

`IF_DEQUEUE(ifq, m)`
This places a pointer to the packet at the head of queue *ifq* in *m* and removes the packet from the queue. A zero value will be returned in *m* if the queue is empty.

`IF_DEQUEUEIF(ifq, m, ifp)`
Like `IF_DEQUEUE`, this removes the next packet from the head of a queue and returns it in *m*. A pointer to the interface on which the packet was received is placed in *ifp*, a `(struct ifnet *)`.

`IF_PREPEND(ifq, m)`
This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro `IF_QFULL(ifq)` returns 1 if the queue is filled, in which case the macro `IF_DROP(ifq)` should be used to increment the count of the number of packets dropped, and the offending packet is dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);
} else
    IF_ENQUEUE(inq, m);
```

12.6. Gateways and Routing Issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in the *Buffering, Congestion Control* section below.

Routing Tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long  rt_hash;                /* hash key for lookups */
        struct  sockaddr rt_dst;        /* destination net or host */
        struct  sockaddr rt_gateway;    /* forwarding agent */
        short   rt_flags;               /* see below */
        short   rt_refcnt;              /* # of references to structure */
        u_long  rt_use;                 /* packets sent using route */
        struct  ifnet *rt_ifp;          /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links.

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). The first appropriate route discovered is used. By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to a "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (the desired final destination), a gateway to which to send the packet, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept, along with a count of "held references" to the dynamically allocated structure to insure that memory reclamation occurs only when the route is not in use. Finally, a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the `rt_dst` field during lookup. If the route is to a network, only a packet's destination network is compared to the `rt_dst` entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will contain the address of the eventual destination, while the local network header will address the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The `RTF_GATEWAY` flag indicates that the route is to an

"indirect" gateway agent, and that the local network header should be filled in from the `rt_gateway` field instead of from the final internetwork destination address.

It is assumed that multiple routes to the same destination will not be present; only one of multiple routes, that most recently installed, will be used.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Current connections may be rerouted after notification of the protocols by means of their `pr_ctlinput()` entries. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using `.netstat(1)`

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

Routing Table Interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine `rtalloc()` performs route allocation; it is called with a pointer to the following structure containing the desired destination:

```
struct route {
    struct  rtentry *ro_rt;
    struct  sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until released with an `rtfree()` call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, allocate and free routes whenever their destination address changes.

The routine `rtredirect()` is called to process a routing redirect control message. It is called with a destination address, the new gateway to that destination, and the source of the redirect. Redirects are accepted only from the current router for the destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accessible from the host are ignored.

User Level Routing Policies

Routing policies implemented in user processes manipulate the kernel routing tables through two `ioctl()` calls. The commands `SIOCADDRT` and `SIOCDELRT` add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies that routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

Several routing policy processes have already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up-to-date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet EGP (Exterior Gateway Protocol), has been accomplished using a similar process.

12.7. Raw Sockets

A raw socket is an object that allows users direct access to a lower-level protocol. Raw sockets are intended for knowledgeable processes that wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

Control Blocks

Every raw socket has a protocol control block of the following form:

```
struct rawcb {
    struct    rawcb *rcb_next;      /* doubly linked list */
    struct    rawcb *rcb_prev;
    struct    socket *rcb_socket;   /* back pointer to socket */
    struct    sockaddr rcb_faddr;   /* destination address */
    struct    sockaddr rcb_laddr;   /* socket's address */
    struct    sockproto rcb_proto;  /* protocol family, protocol */
    caddr_t   rcb_pcb;              /* protocol specific stuff */
    struct    mbuf *rcb_options;    /* protocol specific options */
    struct    route rcb_route;      /* routing information */
    int       rcb_cc;               /*bytes of rawintr queued data*/
    int       rcb_mbcnt;            /*bytes of rawintr queued mbufs*/
    short     rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The `rcb_proto` structure contains the protocol family and protocol number with

which the raw socket is associated. The protocol, family, and addresses are used to filter packets on input; this will be described in more detail shortly. If any protocol-specific information is required, it may be attached to the control block using the `rcb_pcb` field. Protocol-specific options for transmission in outgoing packets may be stored in `rcb_options`. `rcb_cc` and `rcb_mbcnt` are used to keep track of the resources consumed by the raw socket.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, `RAW_LADDR` and `RAW_FADDR`, indicate if a local and foreign address are present. Routing is expected to be performed by the underlying protocol if necessary.

Input Processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives unassigned packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
    struct mbuf *m;
    struct sockproto *proto;
    struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
    struct sockproto raw_proto;
    struct sockaddr raw_dst;
    struct sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form that may be "block compared".

Output Processing

On output the raw `pr_usrreq()` routine passes the packet and a pointer to the raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12.8. Buffering, Congestion Control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in the *Memory, Addressing* section, above. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

Memory Management

The basic memory allocation routines manage a private page map, the size of which determines the maximum amount of memory that may be allocated by the network. A small amount of memory is allocated at boot time to initialize the mbuf and mbuf cluster free lists. When the free lists are exhausted, more memory is requested from the system memory allocator if space remains in the map. If memory cannot be allocated, callers may block awaiting free memory, or the failure may be reflected to the caller immediately. The allocator will not block awaiting free map entries, however, as exhaustion of the resource map usually indicates that buffers have been lost due to a "leak." An array of reference counts parallels the cluster pool and is used when multiple references to a cluster are present.

64 mbufs fit into a 8Kbyte page of memory. Data can be placed into a mbuf by copying, or, better, the memory that contains that data can be treated as a temporary ("loaned") mbuf. This second alternative is far more efficient than an actual copy.

Protocol Buffering Policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines.

Protocols that provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

Queue Limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

Packet Forwarding

When packets can not be forwarded because of memory limitations, the system attempts to generate a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

12.9. Out of Band Data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocol's prerogative to support larger-sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and is usually not stored in the socket's receive queue. A socket-level option, `SO_OOBINLINE`, is provided to force out-of-band data to be placed in the normal receive queue when urgent data is received; this sometimes amelioriates problems due to loss of data when multiple out-of-band segments are received before the first has been passed to the user. The `PRU_SENDOOB` and `PRU_RCVOOB` requests to the `pr_usrreq()` routine are used in sending and receiving data.

12.10. Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of the UNIX system [Gurwitz81].

12.11. References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture*. Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; Specification for the Interconnection of Host and IMP. BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP, RFC-813. Network Information Center, SRI International. July 1982.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection – Basic Reference Model*. ISO/TC 97/SC 16 N 719. August 1981.

[Joy86] Joy, W.; Fabry, R.; Leffler, S.; McKusick, M.; and Karels, M.; Berkeley Software Architecture Manual, 4.3BSD Edition. *UNIX Programmer's Supplementary Documents*, Vol. 1 (PS1:6). Computer Systems Research Group, University of California, Berkeley. May, 1986.

[Leffler84] Leffler, S.J. and Karels, M.J.; Trailer Encapsulations, RFC-893. Network Information Center, SRI International. April 1984.

[Postel80] Postel, J. User Datagram Protocol, RFC-768. Network Information Center, SRI International. May 1980.

[Postel81a] Postel, J., ed. Internet Protocol, RFC-791. Network Information Center, SRI International. September 1981.

[Postel81b] Postel, J., ed. Transmission Control Protocol, RFC-793. Network Information Center, SRI International. September 1981.

[Postel81c] Postel, J. Internet Control Message Protocol, RFC-792. Network Information Center, SRI International. September 1981.

[Xerox81] Xerox Corporation. *Internet Transport Protocols*. Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. Com-28(4); 425-432. April 1980.

Index

M

N

S

T

U

V

W

X

Y

Notes

Notes

Notes